UNIVERSIDAD AUTÓNOMA DE MADRID
ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Desarrollo de la librería scikit-rmt en Python para la simulación y el análisis de matrices aleatorias

Autor: Alejandro Santorum Varela
Tutor: Alberto Suárez González

mayo 2021

Alejandro Santorum Varela
Desarrollo de la librería scikit-rmt en Python
para la simulación y el análisis de matrices aleatorias

Alejandro Santorum Varela

IMPRESO EN ESPAÑA – PRINTED IN SPAIN

Agradecimientos

En primer lugar quiero agradecer a Alberto Suárez, no solo por su implicación y consejos a lo largo de este trabajo, sino por todo lo que me ha enseñado a lo largo de estos cinco años. También quiero mostrar mi agradecimiento a José Luis Fernández y a José Luis Torrea, que han conseguido que aprenda algo de matemáticas.

Por otro lado, quiero agradecer a mis amigos, sin su ayuda y apoyo no sé donde estaría.

Y por supuesto, gracias a mi familia, por **todo**.

Resumen

El interés por el análisis de datos orientado a objetos [1], que es la rama de la estadística que estudia las propiedades de muestras aleatorias de objetos complejos (como funciones, grafos o tensores), está creciendo tanto por los avances en la teoría estadística para el análisis de dicho tipo de datos como por su utilidad en diversos campos de aplicación, como la Economía, la Física o las ingenierías. Las mejoras en la capacidad de cómputo permiten simular, procesar y analizar muestras de gran tamaño de esta clase de objetos de manera eficiente.

En particular, una matriz aleatoria es un tensor bidimensional; es decir, sus elementos están dispuestos en un arreglo de filas y columnas. La Teoría de Matrices Aleatorias, a la que en este documento nos referiremos en ocasiones como RMT (*Random Matrix Theory*), es la rama de la estadística que estudia las propiedades de las matrices cuyas entradas son variables aleatorias. Algunos tipos de matrices aleatorias pueden ser analizados de manera conjunta ya que poseen características comunes, como la distribución de sus elementos o la densidad de sus autovalores. Estas familias de matrices se conocen como *ensembles*.

El objetivo de este trabajo es diseñar e implementar la librería `scikit-rmt` para la simulación, caracterización, procesamiento y análisis de matrices aleatorias. En concreto, se proporcionan herramientas para el muestreo de matrices aleatorias, análisis de su distribución de autovalores, y la ilustración de leyes universales que describen el comportamiento de dichos autovalores. Se desarrollarán asimismo métodos de estimación robusta de matrices de covarianza a partir de datos.

La librería está integrada en el ecosistema *Scipy* de módulos Python para matemáticas, ciencia e ingeniería. Asimismo, el paquete `scikit-rmt` está disponible en el índice oficial de paquetes de Python (PyPI) en https://pypi.org/project/scikit-rmt/ [2], y su documentación es accesible desde la página *Read the Docs* https://scikit-rmt.readthedocs.io/en/latest/ [3].

Palabras clave

Teoría de Matrices Aleatorias (RMT), simulación y muestreo de matrices aleatorias, análisis espectral, estimación de matrices de covarianza, Ley de Wigner, Ley de Marchenko-Pastur, Ley de Tracy-Widom

Abstract

Interest in object-oriented data analysis [1], which is the branch of statistics that studies the properties of random samples of complex objects (such as functions, graphs or tensors), is growing both due to advances in the statistical theory for the analysis of such data as for its usefulness in several fields of application, such us economics, physics or engineering. Improvements in computational power allow the efficient simulation, processing, and analysis of large samples of this class of objects.

In particular, a random matrix is a two-dimensional tensor; that is, its elements are organized in an arrangement of rows and columns. Random Matrix Theory, which in this document we will sometimes refer to as RMT, is the branch of statistics that studies the properties of matrices whose inputs are random variables. Some types of random matrices can be analyzed together since they have common properties, such as the distribution of their elements or the density of their eigenvalues. These families of matrices are known as ensembles.

The goal of this project is to design and implement the `scikit-rmt` library for the simulation, characterization, processing and analysis of random matrices. Specifically, tools are provided for the sampling of random matrices, analysis of their eigenvalue distribution, and the illustration of universal laws that describe the behavior of said eigenvalues. Robust estimation methods of covariance matrices from data will also be developed.

The library is integrated into the *Scipy* ecosystem of Python modules for math, science and engineering. Also, `scikit-rmt` package is available from the official Python Package Index (PyPI) at https://pypi.org/project/scikit-rmt/ [2], and its documentation is accessible from the page *Read the Docs* https://scikit-rmt.readthedocs.io/en/latest/ [3].

Keywords

Random Matrix Theory (RMT), simulation and sampling of random matrices, spectral analysis, covariance matrix estimation, Wigner's Law, Machenko-Pastur Law, Tracy-Widom Law

ÍNDICE

Listas

Lista de algoritmos

Lista de códigos

Lista de ecuaciones

Lista de figuras

Lista de tablas

1

Introducción

El interés por la caracterización de objetos matemáticos aleatorios, como funciones [4], grafos [5] o tensores [6], está creciendo por sus aplicaciones en Economía [7], Física [8] e Ingeniería [9]. En particular, los tensores aleatorios bidimensionales son matrices aleatorias. La Teoría de Matrices Aleatorias o Random Matrix Theory (RMT) es la rama de la estadística que estudia las propiedades de las matrices cuyas entradas son variables aleatorias. Algunas de estas matrices aleatorias tienen propiedades estadísticas comunes (e.g. la distribución de sus autovalores en el límite en el que el tamaño de la matriz es infinito) que son de interés en distintos campos de aplicación. Por esta razón, estos tipos matrices son agrupadas en *ensembles* (conjuntos) para la caracterización de sus propiedades comunes.

La Teoría de Matrices Aleatorias tiene aplicaciones en diversos campos. Por ejemplo, en Estadística se utiliza para generalizar el test χ^2 a varias dimensiones [10]. También es utilizada en Análisis Matemático, para describir errores de cálculo en operaciones con matrices [11]; y en Teoría de Números, ya que la distribución de ceros de la función Zeta de Riemann es modelizada mediante la distribución espectral de ciertas matrices aleatorias [12]. Además, en Física permite modelizar los niveles de energía de los núcleos de átomos pesados [8] y [13]. En Inteligencia Artificial se usa para comprender el rendimiento de una red neuronal a través de las matrices de pesos [9]; y en Finanzas para la modelización de riesgos [14] o para seleccionar carteras de inversión mediante estimación de matrices de covarianza [7]. Finalmente, en Neurociencia se utiliza para describir la red de conexiones sinápticas entre neuronas en el cerebro [15].

1.1. Motivación, objetivos y alcance

Dado el gran número y la importancia de las aplicaciones en las que las matrices aleatorias juegan una papel significativo, es necesario disponer de herramientas computacionales para la simulación, caracterización, manipulación y análisis des este tipo de datos. Las librerías que han sido desarrolladas hasta la fecha proporcionan una funcionalidad insuficiente para realizar un análisis completo de este tipo de datos. El objetivo de este trabajo es desarrollar la librería `scikit-rmt` [16] para completar, al menos parcialmente, tales carencias. La librería está integrada en *SciPy*, un ecosistema de software abierto para matemáticas, ciencia e ingeniería, desarrollado en Python [17].

Entre los objetivos específicos se encuentra la simulación eficiente de las aleatorias de los distintos tipos de *ensembles*. Muchas de las aplicaciones de las matrices aleatorias se basan en el estudio de las características de su espectro de autovalores. Por esta razón, uno de nuestros principales objetivos es implementar herramientas para la representación gráfica de su distribución de autovalores. La simulación de matrices aleatorias y su espectro puede ser acelerada mediante las técnicas detalladas por Alan Edelman en [18] y en [19]. Adicionalmente, la librería se amplía incorporando métodos de estimación de matrices de covarianza propuestos por Oliver Ledoit y Michael Wolf en [20] y [21].

El software desarrollado es de código abierto y escrito en Python, debido a su versátilidad y a su creciente uso en el sector científico e industrial. Está disponible para la audiencia en el repositorio PyPI (*Python Package Index*) [22] y documentado en *Read the Docs* [23]. La librería es desarrollada siguiendo el estilo de SciKits [24] para el desarrollo de paquetes científicos de código abierto en Python aprobados y licenciados por la Open Source Initiative (OSI) [25]. La nomenclatura coherente con el estilo SciKits, `scikit-rmt`, permite a nuestro paquete ser descubierto fácilmente por la comunidad, en lugar de ser uno de los "más de 30000 paquetes de Python no relacionados con la investigación" [24]. Ejemplos de otras librerías que siguen el estilo de SciKits son `scikit-learn` [26], uno de los paquetes de aprendizaje automático más usados, y `scikit-fda` [27], desarrollado por el Grupo de Aprendizaje Automático de la UAM. Por último, para asegurar un producto de alta calidad, seguimos rigurosos estándares de programación como PEP 8 para el estilo de código [28] y PEP 257 para el estilo de documentación [29].

1.2. Estructura de este documento

En el capítulo 2 se realiza una breve introducción a la Teoría de Matrices Aleatorias, en la que se definen los principales *ensembles* de matrices aleatorias, sus principales propiedades y algunas de sus aplicaciones. En el capítulo 3 se realiza un análisis comparativo de las características y funcionalidad de las principales librerías desarrolladas para el tratamiento de matrices aleatorias.

El capítulo 4 detalla el desarrollo de la librería. En concreto, se describen las fases de análisis, de diseño y de implementación. Se exponen las decisiones técnicas tomadas, así como las metodologías y estándares seguidos. Además, en el capítulo 5, las funcionalidades principales que proporciona la librería son ilustradas mediante simulaciones y ejemplos de uso. Finalmente, en el capítulo 6 se exponen las conclusiones, así como las posibles mejoras y desarrollos para un trabajo futuro.

Al final de este documento se encuentran los apéndices. El apéndice A describe el proceso de tridiagonalización de una matriz simétrica. En el apéndice B se muestra cómo podemos muestrar en Python todas las matrices aleatorias consideradas. En C añadimos el código para realizar ciertas simulaciones de construcción de histogramas. Por último, los apéndices D y E muestran ejemplos de uso y simulaciones de la librería `scikit-rmt`.

2

Teoría de Matrices Aleatorias

En este capítulo se realiza una breve introducción a la Teoría de Matrices Aleatorias (RMT). Para ello se revisarán conceptos del álgebra lineal necesarios para establecer las bases de la Teoría de Matrices Aleatorias. A continuación se introducen los *ensembles* de matrices aleatorias más estudiados y utilizados: *Ensembles* Gaussiano, de Wishart, Manova y Circular. Una de las propiedades de interés es la distribución de autovalores para cada uno de los *ensembles*. Por este motivo, se estudiarán las propiedades espectrales correspondientes, así como ciertas técnicas descritas en [18] para acelerar el proceso de simulación. Finalmente, veremos la importancia de la RMT en el problema de estimación de matrices de covarianza.

En cuanto a notación, $\mathcal{M}(\mathbb{R})$ y $\mathcal{M}(\mathbb{C})$ son los conjuntos de matrices con entradas reales y complejas respectivamente. Una variable matemática en negrita $\mathbf{M}$ representa una matriz, la cual puede ser conjugada $\overline{\mathbf{M}}$ si es compleja, o traspuesta $\mathbf{M}^T$. La matriz idéntidad de tamaño n se denotará como I_n. Además, un objeto matemático con doble trazado representa un vector, como $\mathbb{X}$.

2.1. Conceptos de Álgebra Lineal

Comenzar definiendo que $\mathbf{M}$ es una **matriz simétrica** si es igual a su transpuesta, es decir, $\mathbf{M} \in \mathcal{M}(\mathbb{R})$ es simétrica si $\mathbf{M} = \mathbf{M}^T$. Análogamente en el cuerpo de los complejos, $\mathbf{M}$ es una **matriz hermítica** si es igual a su transpuesta conjugada; es decir, $\mathbf{M} \in \mathcal{M}(\mathbb{C})$ es hermítica si $\mathbf{M} = \overline{\mathbf{M}}^T$.

Una **matriz ortogonal** (o de rotación) es aquella que su transpuesta coincide con su inversa; es decir, $\mathbf{M} \in \mathcal{M}(\mathbb{R})$ es ortogonal si $\mathbf{M}\mathbf{M}^T = I_n$. En el caso complejo, $\mathbf{M}$ es una **matriz unitaria** si su transpuesta conjugada coincide con su inversa; es decir, $\mathbf{M} \in \mathcal{M}(\mathbb{C})$ es unitaria si $\mathbf{M}\overline{\mathbf{M}}^T = I_n$.

Por otro lado, una matriz $\mathbf{M} \in \mathcal{M}(\mathbb{R})$ es **conjugada ortogonalmente** cuando es multiplicada por una cierta matriz ortogonal $\mathbf{O} \in \mathcal{M}(\mathbb{R})$ de la siguiente forma: $\mathbf{O}\mathbf{M}\mathbf{O}^T$. En el caso de los complejos es similar, si $\mathbf{M} \in \mathcal{M}(\mathbb{C})$ y si $\mathbb{O}$ es una matriz unitaria, entonces $\mathbf{O}\mathbf{M}\mathbf{O}^T$ es **conjugación unitaria**.

Un concepto no tan básico es el de matriz simpléctica. $\mathbf{M} \in \mathcal{M}(\mathbb{R})$ es una **matriz simpléctica** si $\mathbf{M}^T\mathbf{\Omega}\mathbf{M} = \mathbf{\Omega}$, con

$$\mathbf{\Omega} = \begin{pmatrix} \mathbf{0} & \mathbf{I_n} \\ -\mathbf{I_n} & \mathbf{0} \end{pmatrix}$$

Las matrices simplécticas forman el **grupo simpléctico** [30].

2.2. *Ensembles* de matrices aleatorias fundamentales

En esta sección se describen brevemente cuatro de los principales *ensembles* de matrices aleatorias: el *Ensemble* Gaussiano o de Hermite, el *Ensemble* de Wishart o de Laguerre, el *Ensemble* Manova o de Jacobi y el *Ensemble* Circular o de Dyson.

2.2.1. *Ensemble* Gaussiano

Cuando uno escucha por primera vez el concepto de matriz con entradas aleatorias posiblemente piense en una matriz **M** cuadrada $n \times n$ cuyas entradas son normales estándar independientes. Pues bien, eso se acerca bastante al tipo de matriz aleatoria más básico. Como además buscamos que la matriz sea diagonalizable para poder estudiar sus autovalores, le pedimos a **M** que sea simétrica. Así, **M** no sólo será diagonalizable, sino que también todos sus autovalores serán reales.

La simetrización habitual es $\mathbf{M} := (\mathbf{X} + \mathbf{X}^T)/2$, donde **X** es una matriz cuadrada $n \times n$ con entradas aleatorias normales estándar independientes, y nos aseguramos que **M** tenga n autovalores reales.

La matriz **M** generada mediante el procedimiento descrito pertenece al *ensemble* de matrices aleatorias más conocido: el *ensemble* ortogonal gaussiano (GOE, por sus siglas en inglés).

Adicionalmente, podemos generar la matrix **X** con entradas aleatorias normales estándar complejas o cuaterniónicas y, siguiendo el mismo proceso de simetrización, generaríamos matrices aleatorias de los *ensembles* unitario gaussiano y simpléctico gaussiano respectivamente.

Formalizamos lo anterior presentando los tres sub-*ensembles* Gaussianos.

- **Gaussian Orthogonal Ensemble (GOE) o *Ensemble* Ortogonal Gaussiano**:
  El *Ensemble* Ortogonal Gaussiano (*Gaussian Orthogonal Ensemble*) está formado por matrices $n \times n$ reales simétricas invariantes bajo conjugaciones ortogonales. Decimos que $\mathbf{M} \sim \text{GOE}(n)$ si sus entradas de la diagonal son $N(0, 2)_{\mathbb{R}}$, y las entradas fuera de la diagonal $N(0, 1)_{\mathbb{R}}$.

- **Gaussian Unitary Ensemble (GUE) o *Ensemble* Unitario Gaussiano**:
  El *Ensemble* Unitario Gaussiano (*Gaussian Unitary Ensemble*) está formado por matrices $n \times n$ complejas hermíticas invariantes bajo conjugaciones unitarias. Decimos que $\mathbf{M} \sim \text{GUE}(n)$ si sus entradas de la diagonal son $N(0, 1)_{\mathbb{R}}$, y las entradas fuera de la diagonal $N(0, 1)_{\mathbb{C}}$.

- **Gaussian Symplectic Ensemble (GSE) o *Ensemble* Simpléctico Gaussiano**:
  El *Ensemble* Simpléctico Gaussiano (*Gaussian Symplectic Ensemble*) está formado por matrices $2n \times 2n$ cuaterniónicas hermíticas, es decir matrices simétricas compuestas por cuaterniones, invariantes bajo conjugaciones por el grupo simpléctico. Para una breve introducción sobre cuaterniones y sobre el grupo simpléctico se puede visitar [31] y [30] respectivamente.

2.2.2. *Ensemble* de Wishart

Para explicar este conjunto primero vamos a definir las matrices aleatorias de Wishart y su distribución. La distribución de Wishart surge como una generalización matricial de la distribución χ^2. Si $X_1, X_2, ..., X_n$ son variables aleatorias independientes con $X_i \sim N(0,1)$ entonces $X_1^2+X_2^2+...+X_n^2 \sim \chi_n^2$, donde n indica que la distribución χ^2 tiene n grados de libertad. Cuando los $\mathbb{X}_i$ son vectores aleatorios en lugar de variables aleatorias, digamos $\mathbb{X}_i \sim N_p(\mathbf{0}, I_p)$ con valores en $\mathbb{R}^p$, una posible generalización de la suma de cuadrados anterior es formar la matriz $p \times p$ semidefinida positiva $\mathbf{M} = \sum_{i=1}^{n} \mathbb{X}_i \mathbb{X}_i^T$.

De forma más general, la distribución de Wishart es una familia de distribuciones de probabilidad definidas sobre las matrices simétricas semidefinidas positivas. Sea $\mathbf{X}$ una matriz $p \times n$ cuyas columnas $\mathbf{X} = [\mathbb{X}_1, ..., \mathbb{X}_n]$ son vectores aleatorios p-dimensionales tales que $\mathbb{X}_i \sim N_p(\mathbf{0}, I_p)$, entonces la matriz $\mathbf{M} = \mathbf{X}\mathbf{X}^T = \sum_{i=1}^{n} \mathbb{X}_i \mathbb{X}_i^T$ es una matriz de Wishart.

Si ahora permitimos a los vectores que conforman $\mathbf{X}$ pertenecer al cuerpo de los números complejos o al cuerpo de cuaterniones, estaríamos definiendo otro tipo de matrices aleatorias con propiedades similares a las matrices de Wishart anteriores. Igual que en el caso Gaussiano, podemos considerar los siguientes tres sub-*ensembles* de Wishart.

- **Wishart Real Ensemble (WRE) o *Ensemble* Real de Wishart**:
  El *Ensemble* Real de Wishart (*Wishart Real Ensemble*) está formado por matrices $p \times p$ reales simétricas semidefinidas positivas tales que $\mathbf{M} = \mathbf{X}\mathbf{X}^T = \sum_{i=1}^{n} \mathbb{X}_i \mathbb{X}_i^T$ con $\mathbb{X}_i \sim N_p(\mathbf{0}, I_p)_{\mathbb{R}}$.
- **Wishart Complex Ensemble (WCE) o *Ensemble* Complejo de Wishart**:
  El *Ensemble* Complejo de Wishart (*Complex Wishart Ensemble*) está formado por matrices $p \times p$ complejas hermíticas tales que $\mathbf{M} = \mathbf{X}\mathbf{X}^T = \sum_{i=1}^{n} \mathbb{X}_i \mathbb{X}_i^T$ con $\mathbb{X}_i \sim N_p(\mathbf{0}, I_p)_{\mathbb{C}}$.
- **Wishart Quaternion Ensemble (WQE) o *Ensemble* Cuaterniónico de Wishart**:
  El *Ensemble* de Cuaterniones de Wishart (*Quaternion Wishart Ensemble*) está formado por matrices $2p \times 2p$ complejas hermíticas tales que $\mathbf{M} = \mathbf{A}\mathbf{A}^T$, con $\mathbf{A}$:

  $$\mathbf{A} = \begin{pmatrix} \mathbf{X} & \mathbf{Y} \\ -\overline{\mathbf{Y}} & \overline{\mathbf{X}} \end{pmatrix}$$

  donde $\mathbf{X}$ e $\mathbf{Y}$ son matrices formadas por vectores columna distribuidos como $N_p(\mathbf{0}, I_p)_{\mathbb{C}}$. Las entradas de este tipo de matrices pertenecen al grupo de cuaterniones.

2.2.3. *Ensemble* Manova

Las matrices del *ensemble* Manova pueden ser consideradas como un tipo de 'matrices dobles' de Wishart. El análisis de los tres tipos de sub-*ensemble* permitirá entender mejor la pertinencia de esta

caracterización informal.

Analizando los tres sub-*ensembles* Manova lo entenderemos mejor.

- **Manova Real Ensemble (MRE) o *Ensemble* Real Manova**:
  El *Ensemble* Real Manova (*Real Manova Ensemble*) está formado de la siguiente forma: sea $\mathbf{A_1} \sim \text{WishartReal}(p, n1)$, $\mathbf{A_2} \sim \text{WishartReal}(p, n2)$ y $\mathbf{A} = \mathbf{A_1} + \mathbf{A_2}$, entonces $\mathbf{M} = \mathbf{A_1} \cdot \mathbf{A}^{-1}$ es una matriz aleatoria del *ensemble* Manova Real.

- **Manova Complex Ensemble (MCE) o *Ensemble* Complejo Manova**:
  Siguiendo la misma idea que su versión real, el *Ensemble* Complejo Manova (*Complex Manova Ensemble*) se genera como sigue: sea $\mathbf{A_1} \sim \text{WishartComplex}(p, n1)$, $\mathbf{A_2} \sim \text{WishartComplex}(p, n2)$ y $\mathbf{A} = \mathbf{A_1} + \mathbf{A_2}$, entonces $\mathbf{M} = \mathbf{A_1} \cdot \mathbf{A}^{-1}$ es una matriz aleatoria del *ensemble* Manova Complejo.

- **Manova Quaternion Ensemble (MQE) o *Ensemble* Cuaterniónico Manova**:
  De igual forma, el *Ensemble* Cuaterniónico Manova (*Quaternion Manova Ensemble*) se muestrea así: sea $\mathbf{A_1} \sim \text{WishartQuaternion}(p, n1)$, $\mathbf{A_2} \sim \text{WishartQuaternion}(p, n2)$ y $\mathbf{A} = \mathbf{A_1} + \mathbf{A_2}$, entonces $\mathbf{M} = \mathbf{A_1} \cdot \mathbf{A}^{-1}$ es una matriz aleatoria del *ensemble* Manova de Cuaterniones.

2.2.4. *Ensemble* Circular

Las matrices del *Ensemble* Circular fueron introducidas por Freeman Dyson [32] como modificaciones de las matrices del *Ensemble* Gaussiano con el objetivo de modelar sistemas físicos complejos. Los *ensembles* circulares son medidas en espacios de medidas unitarias y tienen la peculiaridad que todos sus autovalores se sitúan en el círculo unidad complejo.

Los tres *ensembles* de Dyson son los siguientes.

- **Circular Unitary Ensemble (CUE) o *Ensemble* Unitario Circular**:
  Las matrices del *Ensemble* Unitario Circular (*Circular Unitary Ensemble*) conforman el grupo de todas las matrices $n \times n$ unitarias $\mathbb{U}(n)$ dotadas de la medida de Haar (que proviene de la estructura grupal de $\mathbb{U}(n)$).

- **Circular Orthogonal Ensemble (COE) o *Ensemble* Ortogonal Circular**:
  Las matrices del *Ensemble* Ortogonal Circular (*Circular Orthogonal Ensemble*) componen el conjunto de matrices reales simétricas $n \times n$ dotadas de la medida de Haar mediante el mapeo
  $$\mathbb{U}(n) \to \text{COE}(n)$$
  $$\mathbf{U} \mapsto \mathbf{U^T U}$$

- **Circular Symplectic Ensemble (CSE) o *Ensemble* Simpléctico Circular**:
  El *Ensemble* Simpléctico Circular (*Circular Symplectic Ensemble*) CSE$(2n)$ es el conjunto de matrices complejas cuaterniónicas $2n \times 2n$ que son unitarias y auto-duales, dotadas de la medida

de Haar mediante el mapeo

$$\mathbb{U}(2n) \to \mathsf{CSE}(n)$$
$$\mathbf{U} \mapsto \mathbf{U^R U}$$

$$\text{con } \mathbf{U^R} = \begin{pmatrix} \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} & & \\ & \ddots & \\ & & \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \end{pmatrix} \mathbf{U^T} \begin{pmatrix} \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} & & \\ & \ddots & \\ & & \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \end{pmatrix}$$

2.3. Espectro de matrices aleatorias

Las propiedades más estudiadas de las matrices aleatorias de los *ensembles* anteriores tienen que ver con la distribución de sus autovalores, ya que estos codifican gran parte de la información de las mismas. Como consecuencia, una de las herramientas mas útiles es la representación gráfica de su espectro de autovalores.

En la subsección 2.3.1 introduciremos algunas de las Leyes que describen las características asintóticas de las matrices aleatorias descritas en la sección 2.2. Además, en la subsección 2.3.2 se describe el método habitual de construcción de histogramas de autovalores a partir del conjunto de autovalores de una matriz aleatoria. En la subsección 2.3.3 se utilizan ciertas propiedades de los *ensembles* Gaussiano y de Wishart para computar su histograma espectral de forma eficiente.

2.3.1. Leyes espectrales de matrices aleatorias

Las matrices aleatorias se agrupan en *ensembles*, tal y como se expuso en 2.2, debido sus similares características. Una de ellas es que su espectro de autovalores tiene el mismo comportamiento límite, es decir, cuando el tamaño de la matriz aleatoria tiende a infinito sus autovalores siguen una cierta distribución dependiendo del *ensemble*. La distribución límite espectral es descrita mediante Leyes, que caracterizan el comportamiento universal para todas las matrices aleatorias de un mismo *ensemble*. La caracterización de las propiedades espectrales no requiere el uso de diferentes realizaciones de las matrices aleatorias: los correspondientes estadísticos pueden ser estimados a partir de matrices de tamaño suficientemente grande. En la práctica se observa que el régimen asintótico en el que las estimaciones son fiables (los errores muestrales son pequeños en relación con la cantidad estimada) se alcanza con tamaños relativamente pequeños (del orden de matrices 1000×1000).

La primera ley que mencionaremos es posiblemente la más conocida en la literatura de la Teoría de Matrices Aleatorias, la Ley Semicircular de Wigner. Existe una distribución de probabilidad universal μ_{sc} tal que la densidad de los autovalores de cualquier matriz de Wigner (con segundo momento ρ)

converge a ella. En particular, las matrices del *Ensemble* Gaussiano son matrices de Wigner. Esta distribución límite es conocida como la **Ley Semicircular de Wigner**.

Definición 2.3.1 (Ley Semicircular de Wigner)**.** *La distribución de probabilidad μ_{sc} con parámetro ρ se conoce como la Ley Semicircular de Wigner:*

$$\mu_{sc} = \frac{1}{2\pi\rho}\sqrt{4\rho - x^2}\mathbf{1}_{|x|\leq 2\sqrt{\rho}}dx. \tag{2.1}$$

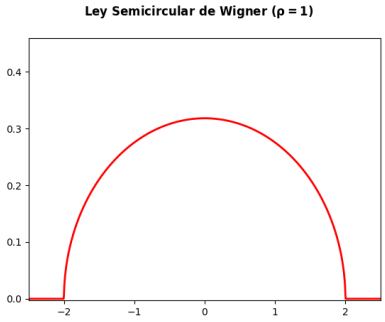


Figura 2.1: Ley Semicircular de Wigner con $\rho = 1$ [1]

Si ahora nos centramos únicamente en el máximo autovalor de una matriz del *Ensemble* Gaussiano, resulta que también es conocida la densidad (determinista) a la que se aproxima.

Definición 2.3.2 (Ley de Tracy-Widom [33])**.** *La **distribución de Tracy-Widom** o **Ley de Tracy-Widom** es la distribución de probabilidad del mayor autovalor de una matriz aleatoria de Wigner. En particular, del Ensemble Gaussiano.*

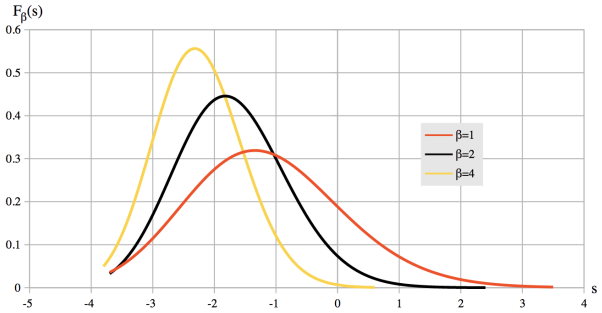


Figura 2.2: Gráfica teórica Ley de Tracy-Widom [2]

En la representación 2.2 podemos ver la función de densidad de probabilidad de Tracy-Widom para las matrices GOE ($\beta = 1$), para las GUE ($\beta = 2$) y para las GSE ($\beta = 4$).

Para mayor información uno puede visitar el artículo original de Tracy y Widom [33].

Por otro lado, también existen resultados para describir el comportamiento límite de los autovalores de las matrices del *Ensemble* de Wishart.

[1] Elaboración propia

[2] Fuente: https://en.wikipedia.org/wiki/Tracy%E2%80%93Widom_distribution

Definición 2.3.3 (Ley de Marchenko-Pastur)**.** *La **distribución de Marchenko-Pastur** o **Ley de Marchenko-Pastur** describe el comportamiento asintótico de los autovalores de una matriz de Wishart. Si el radio $p/n \to \lambda \in (0, \infty)$ cuando $p, n \to \infty$, entonces la distribución espectral de $\mathbf{M}$, que es una función de densidad discreta, converge débilmente con probabilidad 1 a una función determinista y continua conocida como la Ley de Marchenko-Pastur con parámetro λ.*

Si $\lambda \in (0, 1]$, entonces con probabilidad 1 lo siguiente se cumple para cada $x \in \mathbb{R}$:

$$F^{\mathbf{M}}(x) := \frac{1}{p}\#\{1 \leq i \leq p : \lambda_i(\mathbf{M}) \leq x\} \overset{\substack{n,p\to\infty \\ p/n\to\lambda}}{\longrightarrow} \int_{-\infty}^{x} f_\lambda(t)dt \tag{2.2}$$

donde $f_\lambda(t)$ es la función de densidad de probabilidad de Marchenko-Pastur

$$f_\lambda(x) = \frac{1}{2\pi\sigma^2}\frac{\sqrt{(\lambda_+ - x)(x - \lambda_-)}}{\lambda x}, \text{ donde } \lambda_\pm = \sigma^2(1 \pm \sqrt{\lambda})^2, \tag{2.3}$$

y σ^2 es la varianza de las entradas de $\mathbf{X}$, que conforman la matriz de Wishart $\mathbf{M} = \mathbf{X}\mathbf{X}^T$.

Un resultado similar se obtiene si $\lambda > 1$. En este caso la distribución límite tiene masa de probabilidad puntual en el origen cuyo valor es de $1 - \frac{1}{\lambda}$.

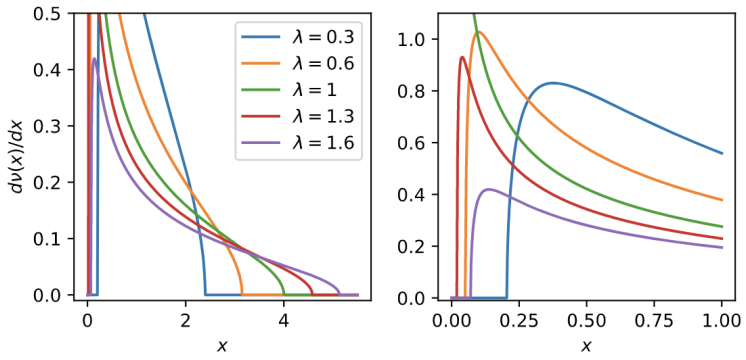


Figura 2.3: Gráfica teórica Ley de Marchenko-Pastur para diferentes ratios λ [3]

La derivación de la función de densidad de probabilidad (2.3), así como pruebas de la Ley de Marchenko-Pastur, se pueden encontrar en el capítulo 3 de [34].

2.3.2. Construcción estándar de histogramas

Para estimar de manera empírica la distribución espectral de una matriz aleatoria dada, es necesario generar una muestra de tamaño suficiente, diagonalizar las matrices generadas y constuir el correspondiente diagrama de autovalores.

Estos histogramas permiten a los investigadores descubrir nuevos fenómenos o ratificar los resul-

[3] Fuente: https://en.wikipedia.org/wiki/Marchenko%E2%80%93Pastur_distribution

tados teóricos probados.

El algoritmo estándar para construir un histograma de autovalores consiste en fijar el intervalo de representación y el número de contenedores (número de *bins*) y calcular los valores propios de la matriz en cuestión. Una vez calculados los valores propios de la matriz en cuestión se determina cuántos de entre ellos están comprendidos en los subintervalos correspondientes a cada uno de los contenedores. Finalmente, una vez calculadas las frecuencias de autovalores en cada caja, se puede generar la gráfica del histograma espectral.

El cálculo de autovalores mediante el algoritmo estándar DSTEQR de LAPACK [35] (el usado por la mayoría de librerías) es la parte más costosa de los cómputos para el histograma, que tiene un coste computacional de $O(n^2)$, siendo n el tamaño de la matriz aleatoria (el número de autovalores). Además, hay que contar el número de autovalores por cada uno de los m *bins* fijados, lo que requiere de un tiempo extra $O(mn)$ ($O(m+n)$ si estuviesen ya ordenados al calcularlos), resultando en un coste computacional total de $O(m+n^2)$.

2.3.3. Construcción eficiente de histogramas

Para ciertos *ensembles* de matrices aleatorias es posible acelerar la construcción de los histogramas aprovechando ciertas propiedades que estos poseen. La idea central reside en construir los histogramas sin la necesidad de calcular explícitamente los autovalores de las matrices aleatorias.

Veamos algunas de esas propiedades y resultados para utilizarlos a nuestro favor.

Definición 2.3.4. *Sea $\mathbf{M}_\beta$ una matriz del Ensemble Gaussiano. Sea $\beta = 1$ si $\mathbf{M}_\beta \in$ GOE(n), $\beta = 2$ si $\mathbf{M}_\beta \in$ GUE(n) o $\beta = 4$ si $\mathbf{M}_\beta \in$ GSE(n). Entonces decimos que $\mathbf{M}_\beta$ está en su* ***forma tridiagonal*** *si*

$$\mathbf{M}_\beta \sim \frac{1}{\sqrt{2}} \begin{pmatrix} N(0,2) & \chi_{(n-1)\beta} & & & & \\ \chi_{(n-1)\beta} & N(0,2) & \chi_{(n-2)\beta} & & & \\ & \ddots & \ddots & \ddots & & \\ & & \chi_{2\beta} & N(0,2) & \chi_\beta & \\ & & & \chi_\beta & N(0,2) & \end{pmatrix} \tag{2.4}$$

Teorema 2.3.5. *Sea $\mathbf{A} \in$ GOE(n), entonces la reducción de $\mathbf{A}$ a su forma tridiagonal $\mathbf{T}$, muestra que $\mathbf{T}$ tiene la misma función de densidad de probabilidad de autovalores que la matriz original $\mathbf{A}$.*

Demostración. Podemos reescribir $\mathbf{A} = \begin{pmatrix} a_n & x^T \\ x & \mathbf{B} \end{pmatrix}$, donde $a_n \sim N(0,1)$, $x \sim N_{n-1}(\mathbf{0}, \frac{1}{2} I_{n-1})$, y $\mathbf{B} \in$ GOE$(n-1)$. Notemos que a_n, x y $\mathbf{B}$ son independientes entre ellos. Sea $\mathbf{H} \in \mathcal{M}_{n-1}$ ortogonal dependiente de x tal que

$$\mathbf{H}x = (||x||_2, 0, \ldots, 0)^T = ||x||_2 e_1,$$

donde $e_1 = (1, 0, \ldots, 0)^T$. Entonces obtenemos

$$\begin{pmatrix} 1 & 0 \\ 0 & \mathbf{H} \end{pmatrix} \begin{pmatrix} a_n & x^T \\ x & \mathbf{B} \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & \mathbf{H}^T \end{pmatrix} = \begin{pmatrix} a_n & ||x||_2 e_1^T \\ ||x||_2 e_1 & \mathbf{HBH}^T \end{pmatrix}.$$

Notemos de nuevo que a_n, $||x||_2$ y $\mathbf{HBH}^T$ son independientes, y como $\mathbf{A} \in \text{GOE}(n)$ y $\mathbf{H}$ depende sólo de x, podemos hayar su distribución: la entrada a_n no ha sido modificada y sigue siendo una normal estándar. Por otro lado, la longitud de una normal guassiana de media 0 y varianza $1/2$ sigue una distribución $\frac{1}{\sqrt{2}}\chi_{n-1}$. Finalmente, como las matrices *GOE* son invariantes por conjugación ortogonal, $\mathbf{HBH}^T$ es también una matriz de GOE$(n-1)$.

Hemos tridiagonalizado la primera columna y primera fila de $\mathbf{A}$, y procediendo por inducción sobre $\mathbf{HBH}^T$ completamos la construcción tridiagonal.

Debido a que las únicas operaciones que realizamos sobre $\mathbf{A}$ son transformaciones ortogonales, los autovalores no se ven afectados y concluimos la demostración. □

Corolario 2.3.6. *Sea $\mathbf{A} \in GUE(n)$ (análogamente $\mathbf{A} \in GSE(n)$), entonces la reducción de $\mathbf{A}$ a su forma tridiagonal $\mathbf{T}$, muestra que la matriz $\mathbf{T}$ tiene la misma función de densidad de probabilidad de autovalores que la matriz original $\mathbf{A}$.*

De forma parecida, para el *Ensemble* de Wishart tenemos:

Definición 2.3.7. *Sea $\mathbf{M}_\beta$ una matriz del Ensemble de Wishart. Sea $\beta = 1$ si $\mathbf{M}_\beta \in WRE(p, n)$, $\beta = 2$ si $\mathbf{M}_\beta \in WCE(p, n)$ o $\beta = 4$ si $\mathbf{M}_\beta \in WQE(p, n)$. Entonces decimos que $\mathbf{M}_\beta$ está en su **forma tridiagonal** si $\mathbf{M}_\beta = \mathbf{B}_\beta \mathbf{B}_\beta^T$ con*

$$\mathbf{B}_\beta \sim \begin{pmatrix} \begin{pmatrix} \chi_{2a} & & & \\ \chi_{(p-1)\beta} & \chi_{2a-\beta} & & \\ & \ddots & \ddots & \\ & & \chi_\beta & \chi_{2a-\beta(p-1)} \end{pmatrix}, \begin{pmatrix} \end{pmatrix} \\ \begin{pmatrix} \end{pmatrix}, \begin{pmatrix} \end{pmatrix} \end{pmatrix} \tag{2.5}$$

para un cierto número real $a > \frac{\beta}{2}(p-1)$.

Teorema 2.3.8. *Sea $\mathbf{G}$ una matriz $p \times n$ con entradas normales estándar i.i.d., entonces $\mathbf{M} = \mathbf{GG}^T$ es una matriz del Ensemble Wishart Real. Reduciendo $\mathbf{G}$ a su forma bidiagonal $\mathbf{B}$ como en (2.5), uno obtiene que la matriz tridiagonal $\mathbf{T} = \mathbf{BB}^T$ con $a = n/2$ tiene la misma función de densidad de probabilidad de autovalores que la matriz $\mathbf{M}$.*

La demostración de este teorema puede encontrarse en la sección 3 de [18].

Corolario 2.3.9. *Análogamente al teorema 2.3.8, si $\mathbf{M}$ es una matriz del Ensemble Complejo de Wishart (respectivamente, del Ensemble Cuaterniónico de Wishart) con parámetro $a = n$ ($a = 2n$), entonces la reducción de $\mathbf{M}$ a su forma tridiagonal $\mathbf{T}$ muestra que ésta última tiene la misma función de densidad de probabilidad de autovalores que la matriz original $\mathbf{M}$.*

Un método de tridiagonalización de matrices simétricas es la **reducción de Householder**, explicada en el apéndice A, así como su algoritmo A.1. No obstante, no buscamos tridiagonalizar las matrices de los *Ensembles* Gaussiano y de Wishart, ya que el propio proceso de tridiagonalización consume el tiempo ahorrado al construir el histograma. Gracias a los teoremas 2.3.5 y 2.3.8, junto con los corolarios 2.3.6 y 2.3.9, podemos muestrar directamente estas matrices aleatorias en su forma tridiagonal, sin tener que tridiagonalizarlas.

En resumen, hemos definido formas más simples de las matrices aleatorias de los *Ensembles* Gaussiano y de Wishart que poseen los mismos autovalores. ¿Qué podemos hacer con las formas tridiagonales de estos *ensembles*? Pues bien, podemos calcular su espectro de autovalores sin tener que calcular los mismos explícitamente. Para conseguir hacer esto, necesitamos antes introducir las *secuencias de Sturm*.

Definición 2.3.10 (Secuencias de Sturm)**.** *Definimos* $(A_0, A_1, \ldots, A_n)$ *a la secuencia de submatrices ancladas a la esquina inferior derecha de la matriz* $\mathbf{A} \in \mathcal{M}_n$*. La* ***secuencia de Sturm*** $(d_0, d_1, \ldots, d_n)_{\mathbf{A}}$ *se define como la secuencia de determinantes* $(|A_0|, |A_1|, \ldots, |A_n|)$.

$$A = A_n = \begin{pmatrix} \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \ldots & a_{nn} \end{pmatrix} \begin{pmatrix} \\ \\ \\ \end{pmatrix} \\ \begin{pmatrix} \\ \\ \\ \end{pmatrix} \begin{pmatrix} \\ \\ \\ \end{pmatrix} \end{pmatrix} \tag{2.6}$$

$$A_1 = \left(\begin{pmatrix} a_{nn} \end{pmatrix} \right), \quad A_2 = \begin{pmatrix} a_{n-1,n-1} & a_{n-1,n} \\ a_{n,n-1} & a_{n,n} \end{pmatrix}, \quad \textit{etc.}$$

El siguiente resultado nos permite utilizar las secuencias de Sturm para hallar el número de autovalores negativos de una cierta matriz.

Lema 2.3.11. *El número de cambios de signo en la secuencia de Sturm* $(d_0, d_1, \ldots, d_n)_{\mathbf{A}}$ *es igual al número de autovalores negativos de* $\mathbf{A}$*.*

La demostración se puede encontrar en la página 228 de [36].

Como en este caso estamos simplemente interesados en el cambio de signo de valores consecutivos, podemos reescribir la secuencia de Sturm $(d_0, d_1, \ldots, d_n)_{\mathbf{A}}$ como la secuencia $(r_1, r_2, \ldots, r_n)_{\mathbf{A}}$, donde $r_i = d_i/d_{i-1}\ \forall i \in \{1, \ldots, n\}$.

Lema 2.3.12. *El número de valores negativos en la secuencia* $(r_1, r_2, \ldots, r_n)_{\mathbf{A}}$ *es igual al número de autovalores negativos de* $\mathbf{A}$*.*

Demostración. Sabemos, por la definición de secuencia de Sturm, que el número de valores negativos en la secuencia $(r_1, r_2, \ldots, r_n)_{\mathbf{A}}$ es igual al número de cambios de signo en $(d_0, d_1, \ldots, d_n)_{\mathbf{A}}$. Por el lema (2.3.11), esto resulta ser igual al número de autovalores negativos de $\mathbf{A}$. □

Como ahora trabajamos con matrices tridiagonales $\mathbf{T}$ de la forma (2.4) en el caso gaussiano o de la forma (2.3.7) en el caso de las matrices de Wishart, con valores $(a_n, a_{n-1}, \ldots, a_1)$ en la diagonal y valores $(b_{n-1}, \ldots, b_1)$ en la super/sub-diagonal, la secuencia de Sturm $(r_1, r_2, \ldots, r_n)_{\mathbf{T}}$ satisface la recurrencia

$$r_i = \begin{cases} a_1, & \text{si } i = 1 \\ a_i - \frac{b_{i-1}^2}{r_{i-1}}, & \text{si } i \in \{2, \ldots, n\} \end{cases} \tag{2.7}$$

Con todo esto en mente, podemos analizar la distribución de los autovalores de una matriz tridiagonal $\mathbf{T}$ del *Ensemble* Gaussiano o de Wishart construyendo su histograma en tiempo $O(mn)$ usando el lema (2.3.12). Como n es habitualmente mucho más grande que m, esto es una mejora significativa sobre el típico procedimiento del algoritmo LAPACK, que involucra calcular los autovalores ($O(n^2)$) y después colocarlos en sus respectivos *bins* ($O(nm)$).

El algoritmo basado en las secuencias de Sturm y en el lema (2.3.12) para la construcción de histogramas es el siguiente: sean $(k_1, \ldots, k_{m-1})$ los separadores entre los contenedores del histograma. Por comodidad, $k_0 = -\infty$ y $k_m = \infty$. El histograma puede ser descrito con la secuencia $(H_1, H_2, \ldots, H_m)$, donde H_i es el número de autovalores entre k_{i-1} y k_i para $1 \leq i \leq m$. Si denotamos $\Lambda(\mathbf{T})$ al número de autovalores negativos de la matriz $\mathbf{T}$, entonces el número de autovalores entre k_1 y k_2 es igual a $\Lambda(\mathbf{T} - k_1 I) - \Lambda(\mathbf{T} - k_2 I)$. Resumiendo:

$$\begin{aligned} H_1 &= \Lambda(\mathbf{T} - k_1 I) \\ H_2 &= \Lambda(\mathbf{T} - k_2 I) - \Lambda(\mathbf{T} - k_1 I) \\ &\vdots \\ H_{m-1} &= \Lambda(\mathbf{T} - k_{m-1} I) - \Lambda(\mathbf{T} - k_{m-2} I) \\ H_m &= n - \Lambda(\mathbf{T} - k_{m-1} I) \end{aligned}$$

Como la función de conteo de autovalores negativos $\Lambda(.)$ tiene complejidad $O(n)$ y hay que evaluarla en m separadores de *bins*, la complejidad global de este procedimiento es $O(mn)$.

En conclusión, es posible reducir el coste computacional de construir los histogramas de las matrices aleatorias de los *Ensembles* Gaussiano y de Wishart de $O(m + n^2)$ a $O(mn)$ si trabajamos con las formas tridiagonales de estos *ensembles*. Esto es una mejora bastante. significativa ya que n suele ser muy grande cuando queremos analizar los comportamientos límite de los autovalores de estas matrices.

Como referencia, los resultados expuestos a lo largo de esta sección, así como las técnicas de aceleración de construcción de histogramas, se pueden encontrar en los artículos de Alan Edelman [18] y [19].

2.4. Estimación de matrices de covarianza

Una de las aplicaciones de la Teoría de Matrices Aleatorias es la estimación de matrices de covarianza. Nuestro objetivo ahora será estudiar algunos de los estimadores de matrices de covarianza a partir de un conjunto de datos (matriz $n \times p$) más utilizados en la actualidad, con aplicaciones en finanzas como, por ejemplo, la construcción de carteras de inversión óptimas propuesta en [7].

2.4.1. Estimadores de matrices de covarianza

Estimador muestral $\mathbf{S}$

Sea $\mathbf{X} = [X_1, ..., X_n]$ es una muestra aleatoria de tamaño n de una población que sigue una distribución $N_p(\boldsymbol{\mu}, \boldsymbol{\Sigma})$. La media muestral $\overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$ y la cuasivarianza muestral $\mathbf{S} = \frac{1}{n-1}\sum_{i=1}^{n}(Xi - \overline{X})(X_i - \overline{X})^T$ son estimadores insesgados de $\boldsymbol{\mu}$ y $\boldsymbol{\Sigma}$ respectivamente.

Notar que $(n-1)\mathbf{S} = \mathbf{Y}\mathbf{Y}^T$ para una cierta matriz $\mathbf{Y}$ cuyas columnas son normales multidimensionales con media cero, es decir, $(n-1)\mathbf{S}$ es una matriz de Wishart.

Además, resulta que $\frac{n-1}{n}\mathbf{S} := \mathbf{S}_n = \frac{1}{n}\sum_{i=1}^{n}(X_i - \overline{X})(X_i - \overline{X})^T$ es el estimador de máxima verosimilitud de $\boldsymbol{\Sigma}$. Es decir, que dependiendo del factor de normalización $\mathbf{S}$ puede ser un estimador insesgado o de máxima verosimilitud. Actualmente es uno de los estimadores de matrices de covarianza más popular y usado en ámbitos como finanzas o aprendizaje automático.

Estimador óptimo de muestra finita (FSOpt o $\mathbf{S}^*$)

El estimador óptimo de muestra finita o *finite-sample optimal estimator* (FSOpt) detallado por Ledoit y Wolf en [37] y en [21] reemplaza los autovalores de la matriz de covarianzas muestral $\mathbf{S}$ por el término (no observable)

$$d_n^* = (d_{n,1}^*, \ldots, d_{n,p}^*) := (u_1^T \boldsymbol{\Sigma} u_1, \ldots, u_p^T \boldsymbol{\Sigma} u_p)$$

antes de recombinarlo con los autovectores de $\mathbf{S}$ para formar

$$\mathbf{S}^* := \sum_{i=1}^{p} d_{n,i}^* u_i u_i^T = \sum_{i=1}^{p} (u_i^T \boldsymbol{\Sigma} u_i) u_i u_i^T \tag{2.8}$$

Subrayar que u_i $i \in \{1, \ldots, p\}$ son los autovectores de la matriz de covarianzas muestral $\mathbf{S}$.

Este estimador tiene el incoveniente de que no es observable, sólo en simulaciones mediante Monte Carlo, ya que se necesita conocer la matriz de covarianzas de la población $\boldsymbol{\Sigma}$, la cual es precisamente la que estamos intentando estimar. El estimador $\mathbf{S}^*$ se utilizará como medida de calidad de estimación para otros métodos, ya que se ha demostrado en [37] y en [21] que minimiza ciertas funciones de coste que veremos próximamente.

Estimador Empírico Bayesiano ($\mathrm{S_{EM}}$)

El estimador empírico bayesiano (*empirical bayesian estimator*) fue introducido por Haff en [38]. Del mismo modo que $\mathbf{S}^*$, $\mathbf{S_{EM}}$ es una combinación lineal de la matriz de covarianzas muestral $\mathbf{S}$ y de la matriz identidad. La diferencia reside en los coeficientes de la combinación: los coeficientes propuestos por Haff no dependen de las observaciones de $\mathbf{X}$, solo en n y p.

$$\mathbf{S^{EM}} := \frac{pn - 2n - 2}{pn^2}\det(\mathbf{S})^{1/p} I_p + \frac{n}{n+1}\mathbf{S}. \tag{2.9}$$

Haff sugerió en 1980 este estimador cuando el criterio de coste fuese el error cuadrático medio.

Estimador Minimax ($\mathrm{S_{MX}}$)

El estimador Minimax fue propuesto en [39]. Es el estimador con menor error en el peor caso. Este estimador es a veces criticado por ser sobre pesimista, debido a que solo se centra en el peor caso.

Se calcula preservando los autovectores de la matriz de covarianzas muestral $\mathbf{S}$ y reemplazando sus autovalores por $\tilde{\lambda}_i := \frac{n}{n+p+1-2i}\lambda_i$, donde λ_i $i \in \{1, \ldots, p\}$ son los autovalores de $\mathbf{S}$ en orden descendiente.

Estimador de contracción lineal

El estimador de contracción lineal (*linear shrinkage estimator*) fue derivado por Ledoit y Wolf en [20].

Para matrices de covarianza de altas dimensiones, el estimador usual muestral $\mathbf{S}$ suele estar mal condicionado, es decir, invertirla puede amplificar el error de estimación. Cuando la dimensión p es mayor que el número de observaciones n, la matriz de covarianzas muestral $\mathbf{S}$ no es invertible. Cuando el ratio p/n es menor que uno pero no despreciable, $\mathbf{S}$ es invertible pero no está bien condicionada.

Para solucionar estos problemas, Ledoit y Wolf propusieron el estimador de contracción lineal, que define un estimador bien condicionado para matrices de covarianza de dimensión alta.

Una forma de obtener un estimador estructurado bien condicionado es imponer la condición de que todas las varianzas son iguales y todas las covarianzas son cero. El estimador de contracción lineal asintóticamente óptimo finalmente recomendado es un promedio ponderado de este estimador estructurado y la matriz de covarianza muestral $\mathbf{S}$.

Estimador analítico de contracción no lineal

En [40] y en [41] se publican nuevos estimadores de matrices de covarianza que mejoraban a los anteriores, conocidos como *NERCOME* y *QuEST* respectivamente. No obstante, Ledoit y Wolf en [21] combinan ambos métodos con el lineal visto anteriormente para crear el estimador analítico de contracción no lineal o *analytical non-linear shrinkage estimator*.

De la rutina *QuEST* se coge la fórmula óptima de contracción no-lineal. Se imita el diseño simple del método *NERCOME*, y la velocidad y escalabilidad del estimador de contracción lineal.

El estimador de contracción no lineal asintóticamente óptimo finalmente propuesto es construido mediante la estimación no parámetrica de la densidad límite espectral de los autovalores de $\mathbf{S}$ y su transformada de Hilbert.

2.4.2. Métricas de calidad de estimación

Una vez que han sido definidos los estimadores a cosiderar durante este trabajo, ahora desearíamos saber cual es mejor en cada caso.

Una de los criterios más usados es la **función de coste de Fröbenius**, definida como

$$\mathcal{L}_n^{FR} := \frac{1}{p}\mathsf{Tr}[(\hat{\mathbf{\Sigma}} - \mathbf{\Sigma})^2] \tag{2.10}$$

La función de coste (2.10) es con respecto a la cual el estimador lineal de Ledoit y Wolf en [20] es optimizado. Además, ambos demostraron en [37] que $\mathbf{S}^*$ es óptimo con respecto a esta métrica.

Por otro lado, el estimador de contracción analítico de Ledoit y Wolf [21] es optimizado con respecto a la siguiente función de coste:

$$\mathcal{L}_n^{MV} := \frac{\mathsf{Tr}(\hat{\mathbf{\Sigma}}^{-1}\mathbf{\Sigma}\hat{\mathbf{\Sigma}}^{-1})/p}{[\mathsf{Tr}(\hat{\mathbf{\Sigma}}^{-1})/p]^2} - \frac{p}{\mathsf{Tr}(\mathbf{\Sigma}^{-1})}, \tag{2.11}$$

conocida como **función de coste de mínima varianza**.

Finalmente, la métrica más representativa para nuestro trabajo será el **PRIAL** (*Percentage relative improvement in average loss*), porcentaje relativo de mejora en coste medio.

$$\text{PRIAL}_n^{MV} := \frac{\mathbb{E}[\mathcal{L}_n^{MV}(\mathbf{S}, \mathbf{\Sigma})] - \mathbb{E}[\mathcal{L}_n^{MV}(\hat{\mathbf{\Sigma}}, \mathbf{\Sigma})]}{\mathbb{E}[\mathcal{L}_n^{MV}(\mathbf{S}, \mathbf{\Sigma})] - \mathbb{E}[\mathcal{L}_n^{MV}(\mathbf{S}^*, \mathbf{\Sigma})]} \times 100\,\%. \tag{2.12}$$

En la práctica, la esperanza $\mathbb{E}(.)$ es tomada como el resultado medio de varias simulaciones Monte Carlo (entre 100 y 500 típicamente).

Por construcción, $\text{PRIAL}_n^{MV}(\mathbf{S}) = 0\,\%$, lo que significa que la matriz de covarianzas muestral representa el punto de referencia a partir del cual la reducción de coste es medida. Un estimador que tenga un coste esperado menor (mayor) que la matriz de covarianza muestral obtendrá un PRIAL positivo (negativo).

También por construcción, $\text{PRIAL}_n^{MV}(\mathbf{S}^*) = 100\,\%$, porque esta es la máxima cantidad de reducción de coste que puede ser conseguida con un estimador por contracción no lineal (Ledoit y Wolf en [21]), ya que recordemos que para construir $\mathbf{S}^*$ necesitábamos conocer la propia matriz $\mathbf{\Sigma}$.

3

Estado del arte

En este capítulo se realizará un análisis de las herramientas computacionales disponibles para la caracterización, análisis y simulación de la Teoría de Matrices Aleatorias.

El objetivo es identificar las funcionalidades que ofrecen dichas herramientas, así como sus ventajas y desventajas. Partiendo de este análisis se realizará una propuesta de funcionalidades y diseño para la herramienta `scikit-rmt`, de forma que se incorporen los elementos positivos y se eviten las limitaciones de las librerías disponibles.

A continuación, se describen brevemente las características de código abierto disponibles en el área de la Teoría de Matrices Aleatorias.

- **EmpyricalRMT** [42]: librería de Python para investigar algunos de los elementos clásicos de la Teoría de Matrices Aleatorias, incluyendo la generación de valores propios y la representación gráfica de sus densidades espectrales.
- **pyRMT** [43]: librería de Python basada en la Teoría de Matrices Aleatorias para estimación de matrices de covarianza con ruido.
- **bristol** [44]: librería de Python para procesamiento paralelo y modelización de matrices aleatorias del *Ensemble* Circular. Incluye asimismo funcionalidades de análisis espectral, como la estimación de la densidad de autovalores de matrices aleatorias.
- **RandomMatrixStability** [45]: librería de R para el análisis de sistemas finitos mediante la Teoría de Matrices Aleatorias.
- **RMAT** [46]: paquete de R para la simulación de *ensembles* de matrices aleatorias. Proporciona también herramientas para la estimación y análisis de sus espectros de autovalores.
- **RandomMatrices** [47]: librería de Julia que proporciona métodos para muestrar y analizar matrices aleatorias y sus distribuciones asociadas. Ha sido implementada por el grupo de Estadística y Aprendizaje Automático en Julia: Julia Statistics.
- **RandomMatrixDistributions** [48]: librería de Julia para muestrar varios *ensembles* de matrices aleatorias y estudiar las leyes universales que siguen sus espectros de autovalores.

En la tabla 3.1 podemos encontrar dicho análisis de la competencia que intenta resumir el estado del arte con respecto al *software* relacionado con la RMT .

Tabla 3.1: Análisis de la competencia

Nombre	Aspectos Positivos	Aspectos Negativos
EmpyricalRMT (Python) [42]	• Construcción eficiente de histogramas mediante muestreo de *ensembles* en forma tridiagonal. • Representación de densidades espectrales incorporando suavizado (*smoothing*). • Aproximación de densidades y distribuciones mediante polinomios. • Uso de estándares (PEP 8).	• Pocos *ensembles* disponibles para su muestreo y análisis: GOE, GUE y Poisson. • Pobre documentación: no todas las funciones están documentadas y sus tutoriales son insuficientes, con pocas explicaciones. •No es una versión estable.
pyRMT (Python) [43]	• Suavizado de matrices de covarianza con varios métodos. • Estimación de matrices de covarianza usando estimadores propuestos por Ledoit y Wolf. • Documentación de código siguiendo el estilo `numpy` de *docstrings*.	• No permite manejar ningún tipo de matriz aleatoria. • Aparentemente no sigue ningún estándar de programación ni realiza *tests* de unidad.
bristol (Python) [44]	• Muestreo de los tres tipos de matrices aleatorias del *Ensemble* Circular: COE, CUE, CSE. • Desarrollo de técnicas de computación paralela. • Representación gráfica de la densidad espectral.	• No permite muestrar otro tipo de matrices aleatorias exceptuando las tres circulares.

RandomMatrix Stability (R) [45]	• Implementa varias aplicaciones de RMT, como análisis de redes aleatorias. • Amplia documentación y de fácil instalación. • Aporta ejemplos de uso con *notebooks*.	• No permite el uso directo con matrices aleatorias de diferentes *ensembles*. • No existen métodos para el análisis del espectro de dichas matrices.
RMAT (R) [46]	• Permite analizar muchos *ensembles* de matrices aleatorias: matrices uniformes, gaussianas, tridiagonales, matrices estocásticas, etc. • Incorpora métodos de estudio del espectro de las anteriores matrices. • Amplia documentación y de fácil instalación.	• Insuficiente número de ejemplos de uso. • Escasos *tests*, lo que puede llevar a un mal funcionamiento de algunas rutinas.
RandomMatrices (Julia) [47]	• Implementación de varios *ensembles* de matrices aleatorias con varias funcionalidad avanzadas, como el Gaussiano, de Wishart, Manova y el Circular. • Simulación de la Ley de Wigner y de Tracy-Widom. • Permite el cálculo de las funciones conjuntas de densidad de los autovalores de los *ensembles* anteriores. • Aporta operadores estocásticos (procesos brownianos).	• *Tests* insuficientes, cubriendo menos del 45 % del código. • Escasa documentación.

RandomMatrix Distributions (Julia) [48]	• Incorpora simulaciones de distribuciones de varias matrices aleatorias: de Wigner, de Wishart y del *Ensemble* de Jacobi. • Simulación de varias leyes: Ley de Marchenko-Pastur, de Tracy-Widom y de Wachter. • Muestreo eficiente usando formas tridiagonales.	• Pobre documentación y tutoriales prácticamente nulos. • *Tests* insuficientes, cubriendo menos del 50 % del código.

Posiblemente la mejor librería de las analizadas anteriormente sea *RandomMatrices* [47]. Esta librería está escrita en Julia, por lo que se espera que sea computacionalmente eficiente, algo muy importante a tener en cuenta en *software* matemático o científico. Adicionalmente, podemos ver que es una librería creada por la propia comunidad de Julia de estadística y aprendizaje automático (Julia Stats), contando con 16 contribuidores expertos en la materia. Fue publicada y está en continuo desarrollo desde el 2013. El número de *ensembles* implementados es considerablemente alto. La funcionalidad soportada es también abundante, permitiendo el análisis de los autovalores de sus *ensembles* y de sus densidades de probabilidad. Además, permite la simulación de varias leyes de RMT : la Ley Semicircular de Wigner y la Ley de Tracy-Widom.

El resto de propuestas se ven eclipsadas por la anterior. Algunas, como la librería *RMAT* [46], propone otros *ensembles* diferentes. No obstante, las librerías de R tienden a ser poco eficientes (a menos que por debajo utilicen rutinas en C) y las librerías de Python, que suelen usar `numpy` (compilada en C) para cálculos numéricos, implementan pocos *ensembles* y/o están escasamente documentadas.

Por lo tanto, nuestra propuesta tomará como referencia el paquete *RandomMatrices* [47] de Julia, ya que implementa una gran cantidad de *ensembles* de matrices aleatorias, permitiendo su muestreo, análisis espectral e incorporación de la tridiagonalización para su muestreo eficiente. Intentaremos mejorar su cobertura de *tests*, que claramente son escasos, para reducir la probabilidad de comportamientos anómalos; así como incluir más ejemplos de uso y una documentación más extensa para facilitar su uso a cualquier individuo de la comunidad científica.

La librería `scikit-rmt` desarrollada incorporará, adicional a lo que aporta *RandomMatrices*, métodos eficaces para mejorar la estimación de matrices de covarianza. Sin embargo, a diferencia con *RandomMatrices*, `scikit-rmt` no incluirá la simulación de procesos estocásticos debido que esta funcionalidad ya la implementa `scikit-fda` [27]; ni herramientas para la manipulación de *Formal Power Series* y *Laurent series* ya que está fuera del alcance del proyecto.

4

Análisis, diseño e implementacion

Recordando los objetivos propuestos, en este proyecto buscamos diseñar e implementar una librería en Python que permita simular y analizar las propiedades de las matrices aleatorias. Para este fin tomaremos como principal base las referencias [18] y [21].

Uno de los subobjetivos es la implementación de métodos de simulación de matrices de los *ensembles* descritos en la sección 2.2. Adicionalmente, se proporcionan herramientas para el análisis de autovalores estas matrices, ya que es en el espectro donde reside la mayor cantidad de información de estos objetos matemáticos. Ampliamos la librería permitiendo mejorar la estimación de matrices de covarianza a partir de una muestra utilizando herramientas de la RMT .

Finalmente, el software desarrollado será código abierto, disponible para la audiencia en el repositorio PyPI (*Python Package Index*) [22], y documentado en Read the Docs [23].

4.1. Análisis de requisitos

Nuestra propuesta se va a dividir en los siguientes subsistemas o módulos:

- **Subsistema de *ensembles* de matrices aleatorias**: el sistema debe permitir al usuario trabajar directamente con matrices aleatorias de diferentes *ensembles*, muestrándolas eficientemente y analizar su espectro, es decir, su conjunto de autovalores.

- **Subsistema de estimación de matrices de covarianza**: el usuario podrá estimar matrices de covarianza utilizando para ello diferentes estimadores. Cada uno de ellos tendrá su punto fuerte (poco sesgo, poca desviación, etc.). Adicionalmente, se podrá medir la calidad de estimación con diferentes métricas de coste.

A continuación detallamos los requisitos funcionales y no funcionales para cada uno de estos módulos.

4.1.1. Requisitos funcionales

Ensembles de matrices aleatorias

RF-1.– Muestreo de matrices aleatorias de los siguientes *ensembles*:

- **RF-1.1.–** *Ensembles* Gaussianos: GOE, GUE, GSE.
- **RF-1.2.–** *Ensemble* de Wishart: WRE, WCE, WQE.
- **RF-1.3.–** *Ensemble* Manova: MRE, MCE, MQE.
- **RF-1.4.–** *Ensembles* Circulares: COE, CUE, CSE.

RF-2.– Muestreo de matrices aleatorias en su forma tridiagonal de los siguientes *ensembles*:

- **RF-2.1.–** *Ensembles* Gaussianos: GOE, GUE, GSE.
- **RF-2.2.–** *Ensembles* de Wishart: WRE, WCE, WQE.

RF-3.– Cálculo de la función conjunta de densidad de probabilidad de los valores propios de las matrices aleatorias muestradas.

RF-4.– Estimación del espectro de autovalores con el procedimiento estándar para todos los *ensembles* anteriores.

RF-5.– Estimación eficiente del espectro de autovalores usando las matrices tridiagonales de los *ensembles* Gaussianos y de Wishart.

RF-6.– Construcción de histogramas que representan el espectro de autovalores para todos los *ensembles* muestrados.

RF-7.– Implementación de simulaciones de las siguientes leyes de RMT:

- **RF-7.1.–** Ley Semicircular de Wigner usando los *ensembles* Gaussianos.
- **RF-7.2.–** Ley de Marchenko-Pastur usando los *ensemgles* de Wishart.
- **RF-7.3.–** Ley de Tracy-Widom usando los *ensembles* Gaussianos.

Matrices de covarianza

RF-7.– Estimación de matrices de covarianza con los siguientes métodos:

- Estimador muestral insesgado de máxima verosimilitud, visto en 2.4.1.
- Estimador óptimo a partir de muestra finita (2.8), descrito en [21].
- Estimador de contracción lineal asintóticamente óptimo visto en 2.4.1 y detallado en [20].
- Estimador de contracción no lineal asintóticamente óptimo en forma analítica visto en 2.4.1 y expuesto en [21].
- Estimador bayesiano empírico (2.9), introducido en [38] y descrito en [20].
- Estimador minimax visto en 2.4.1, introducido en [39].

RF-8.– Medición de la calidad de estimación utilizando las siguientes métricas:

- Función de coste por mínima varianza (2.10), definida en [21].
- Función de coste de Frobenius (2.11), definida en [21].
- Porcentaje de mejora relativa en coste medio (*PRIAL, Percentage relative improvement in average loss*) (2.12), expuesta en [21].

4.1.2. Requisitos no funcionales

Rendimiento y recursos

RNF-1.– Tiempo de respuesta: Para matrices 1000×1000 o menores, el tiempo de ejecución de muestreo deberá ser inferior a 1 segundo. Para el mismo tamaño de matrices, el tiempo de representación de su espectro de autovalores deberá ser inferior a 3 segundos.

RNF-2.– Uso de memoria: el software deberá requerir un uso de memoria RAM proporcional al tamaño (número de entradas) de las matrices aleatorias que se estén usando.

Fiabilidad

RNF-3.– Los *tests* deben cubrir al menos el 90 % del código, exceptuando aquellas rutinas que tengan como salida una representación gráfica.

RNF-4.– La interfaz de las clases, métodos y funciones será estable. Se usarán clases y métodos abstractos o privados para evitar que el usuario utilice código que podría evolucionar.

Extensibilidad

RNF-5.– El *software* deberá ser de fácil modificación y/o extensión por terceros usuarios.

RNF-6.– La librería deberá ser fácilmente ampliable con mayor funcionalidad o con el soporte de más *ensembles* de matrices aleatorias que los implementados.

Portabilidad y compatibilidad

RNF-7.– El paquete será descargable e instalable tanto en Macs como en PC's. Por lo tanto, tendrá que ser apta para Windows 10 o superior, Mac (Sierra 10.12 o superior) y Linux en sus distribuciones Ubuntu 16 o superior y Debian 8 o superior.

Usabilidad

RNF-8.– La librería deberá ser intuitiva y de fácil uso.

RNF-9.– Se aportarán ejemplos de uso y tutoriales con el objetivo de facilitar el comienzo de uso de la librería.

Documentación

RNF-10.– La documentación de la aplicación estará en inglés para poder ser utilizada cualquier usuario.

RNF-11.– La documentación deberá seguir las pautas de estilo aconsejadas por la herramienta Sphinx [49].

RNF-12.– La documentación del software se redactará en el código fuente y se publicará en GitHub [16], en PyPI y en Read the Docs, mediante una herramienta de generación automática de documentación en HTML.

Licencia

RNF-12.– La licencia de la librería deberá ser permisiva en cuanto a su uso privado y modificación, sin garantías y que prohíbe a terceros usar el nombre del proyecto para promover sus productos sin consentimiento por escrito.

RNF-13.– Se permitirá el uso comercial del paquete, siempre y cuando se notifique de antemano al autor del mismo y se cite como en [16]:
A. Santorum, "scikit-rmt", https://github.com/AlejandroSantorum/scikit-rmt, 2021. GitHub repository.

Estilo

RNF-14.– El estilo de código seguirá las guías y directrices de la herramienta *Pylint* [50].

Legales

RNF-15.– Ni el nombre del titular de los derechos de autor ni los nombres de sus colaboradores podrán utilizarse para respaldar o promocionar productos derivados de este *software* sin un permiso previo específico por escrito.

RNF-16.– El *software* es proporcionado ”tal cual” y el autor y colaboradores no se responsabilizan del uso irresponsable del mismo por terceros.

4.2. Diseño

En esta sección vamos a diseñar los módulos considerados en la descomposición en subsistemas en la fase de análisis.

Módulo *ensemble*

Centrándonos primero en el módulo de manejo de *ensembles* de matrices aleatorias, por las características similares que estos poseen decidimos que un diseño orientado a objetos es la mejor elección. En la imagen 4.1 podemos encontrar el diagrama de clases diseñado del módulo *ensemble*.

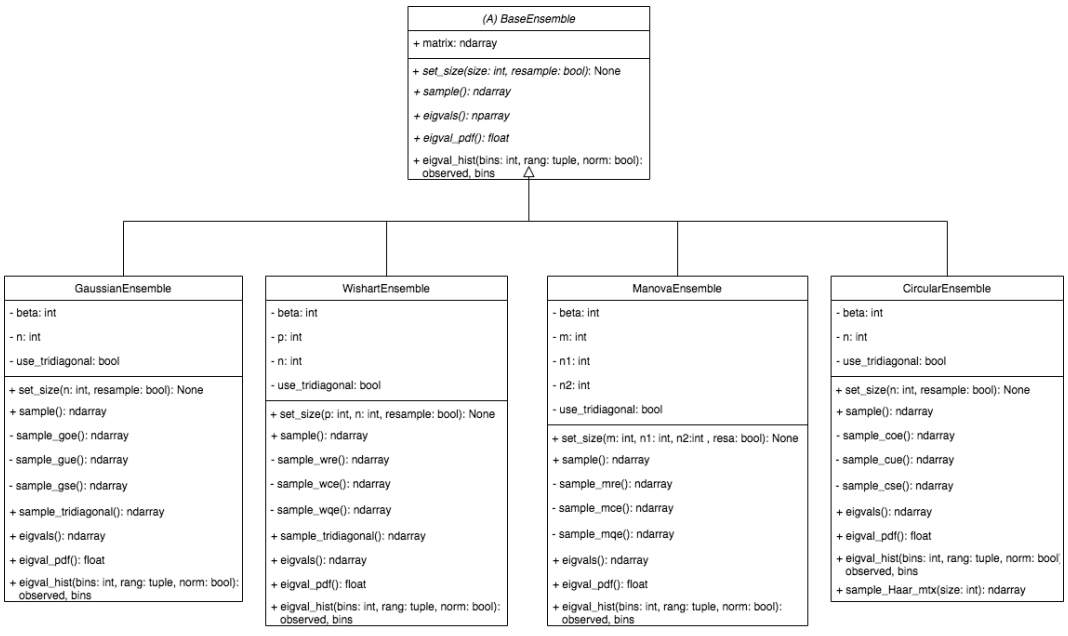


Figura 4.1: Diagrama de clases del módulo *Ensemble*

Como podemos ver, existe una clase padre abstracta `BaseEnsemble` de la que heredan el resto de *Ensembles*. Claramente, esta va a tener un atributo común para todos los *ensembles*, `matrix`, la matriz aleatoria muestrada. En cuanto a métodos, esta clase tiene los que son de común implemen-

tación para todos los *ensembles*, como `eigval_hist`, y especifica aquellos abstractos que tendrán que ser especificados en las clases heradadas: `set_size`, `sample` y `eigval_pdf`.

Heredando la clase anterior se encuentran los cuatro *ensembles* de matrices aleatorias analizados en la sección 2.2, representados con las clases `GaussianEnsemble`, `WishartEnsemble`, `ManovaEnsemble` y `CircularEnsemble`. Cada uno de ellos tiene varios atributos: un número entero `beta` que especifica la instancia del sub-*ensemble* muestrado, el tamaño de la matriz, que dependiendo del *ensemble* necesitará ser especificado de forma diferente: las matrices del *Ensemble* Gaussiano son cuadradas $n \times n$, las matrices de Wishart son formadas a partir de otras de tamaño $p \times n$, las Manova son generadas usando matrices $m \times n_1$ y $m \times n_2$, y las del *Ensemble* Circular son de nuevo $n \times n$. Finalmente, los *Ensembles* Gaussiano y de Wishart tienen un atributo adicional, el *booleano* `use_tridiagonal`, que indica si se desea muestrar las matrices aleatorias en su forma tridiagonal o en su forma estándar.

Todos las clases tendrán métodos públicos para cambiar el tamaño de sus matrices aleatorias (`set_size`), así como un método de muestreo `sample`, que es de obligada especificación al ser heredada de la clase padre. Notar que en las clases `GaussianEnsemble` y `WishartEnsemble` el método `sample` dependerá del atributo `use_tridiagonal` para muestrar las matrices en su forma tridiagonal o estándar. Estas clases tendrán por tanto un método adicional `sample_tridiagonal` que muestrea las matrices aleatorias de su correspondiente *ensemble* en su forma tridiagonal. Por otro lado, todos los *ensembles* tendrán métodos privados para muestrar las distintas matrices aleatorias de sus sub-*ensembles*. Son privadas porque solo serán accesibles desde el método `sample` especificando anteriormente (en el constructor) el parámetro `beta`.

Adicionalmente, todos las clases de los *ensembles* tendrán un método público para el cálculo de sus autovalores, `eigvals`, que dependerá de si la matriz aleatoria de ese *ensemble* es simétrica/hermítica o no. Estos también especificarán el método `eigval_pdf`, que calcula la función conjunta de densidad de los autovalores de las matrices aleatorias. Finalmente, el método `eigval_hist` es especializado en las clases `GaussianEnsemble` y `WishartEnsemble` para construir el histograma eficientemente usando la teoría expuesta en 2.3.3.

Por otro lado, en este submódulo se incluye cierta funcionalidad relevante para el manejo de matrices tridiagonales. Además, se quieren utilizar los *ensembles* de matrices aleatorias para estudiar distintas leyes de autovalores. En la imagen 4.2 representamos el submódulo *Ensemble* al completo. Como comentamos anteriormente, las clases que implementan los distintos *ensembles* de matrices aleatorias siguen un diseño orientado a objetos, pero esto es ampliado con ciertas funciones, que siguen, como es de esperar, un diseño funcional.

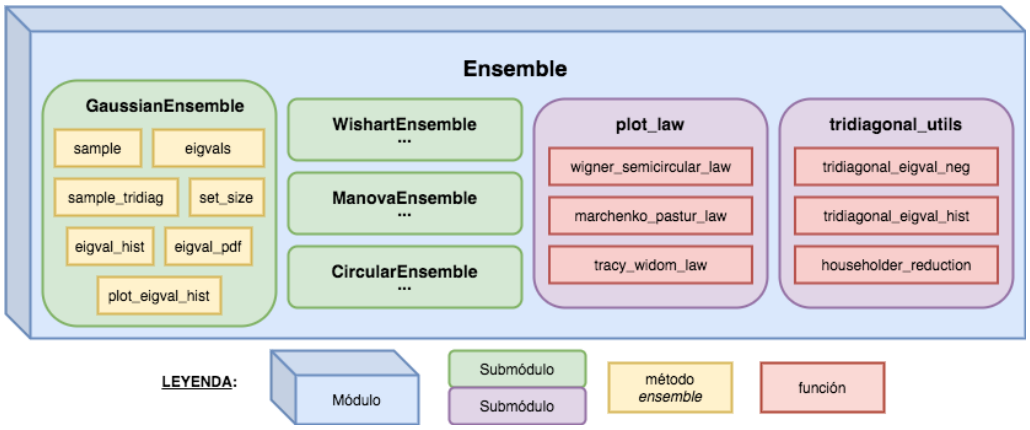


Figura 4.2: Diagrama funcional del módulo *Ensemble*

Las leyes espectrales de matrices aleatorias simuladas por el submódulo *Ensemble* serán la Ley de Wigner, la Ley de Tracy-Widom y la Ley de Marchenko-Pastur, introducidas en 2.3.1.

Se incluyen también funciones para calcular el número de autovalores negativos de una matriz tridiagonal (`tridiagonal_eigval_neg`) y para construir el histograma de autovalores de una matriz tridiagonal dada (`tridiagonal_eigval_hist`). La función `householder_reduction` implementará el método de Householder para reducir una matriz simétrica en su equivalente tridiagonal.

Módulo *covariance*

El módulo de estimación de matrices de covarianza será enfocado de forma diferente. Este subsistema contiene diferentes rutinas que, dada una cierta matriz $\mathbf{X}$ $n \times p$, estas intentan estimar y aproximar la matriz de covarianza real $\mathbf{\Sigma}$ con la que se formó la matriz $\mathbf{X}$.

Como un diseño orientado a objetos no tiene sentido, proponemos un diseño funcional para este módulo.

Las funciones de estimación a implementar son las descritas teóricamente en la sección 2.4. En la figura 4.3 podemos encontrar el diagrama funcional del módulo *covariance*.

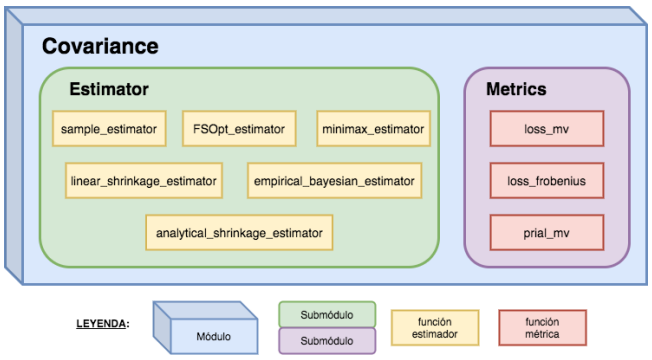


Figura 4.3: Diagrama funcional del módulo *Covariance*

4.3. Implementación

En esta sección se describe de manera detallada los procedimientos seguidos durante la fase de implementación, tales como estilo de programación en Python, control de versiones, documentación, pruebas de unidad y/o integración, etc.

4.3.1. Código

Tal y como se mencionó en 4.1, la librería cuenta con dos módulos: manejo de *ensembles* de matrices aleatorias y estimación de covarianzas. Ambos módulos se apoyan de la librería de cálculo matemático `numpy` [51]. Esta nos permite ejecutar eficientemente funciones de álgebra lineal, generación de distribuciones conocidas y cálculo complejo al estar escrita y compilada en lenguaje C para su uso en Python. El módulo `ensemble` basa su código en la teoría expuesta en las secciones 2.2 y 2.3, así como en los artículos [18], [19] y [52].

El módulo `covariance` se implementa siguiendo las instrucciones de los artículos [20] y [21], cuyos estimadores han sido descritos brevemente en la sección 2.4. Además, para los estimadores lineal y analítico se toma de referencia su código en MatLab, publicado por el propio Michael Wolf en [53].

Por otro lado, se han seguido dos *Python Enhancement Proposals* (PEP), que son dos estándares de programación. En concreto se ha seguido la guía de estilo de programación PEP 8, *Style Guide for Python Code* [28], y la guía de estilo de documentación PEP 257, *Docstring Conventions* [29], que establece directrices para documentar el código.

4.3.2. Control de versiones

Durante el proceso de desarrollo del paquete se ha utilizado el sistema de control de versiones *git* mediante GitHub, que permite almacenar en un servidor remoto los cambios de código que se realizan sobre un proyecto. Además, antes de publicar cualquier actualización a la rama `main` se han ejecutado las pruebas de unidad descritas en la sección 4.3.4, asegurándonos de que los nuevos cambios no interfieren en el funcionamiento anterior de la librería.

Adicionalmente, se enlaza el repositorio de GitHub *Travis CI* [54], que es una herramienta de integración continua. Su objetivo es conseguir detectar errores en el proyecto lo antes posible, realizando pruebas de compilación y corriendo los *tests* cada vez que se realiza alguna modificación.

4.3.3. Documentación

La documentación ha de ser extensiva y completa al ser un proyecto de código abierto, permitiendo a otros desarrolladores utilizar, mejorar o aumentar el software final publicado. Esta sigue el estricto estándar PEP 257, *Docstring Conventions* [29]. Las *Docstrings* que siguen este estándar son usadas para generar páginas HTML o PDFs conteniendo la documentación escrita en el código. En nuestro caso, la herramienta de documentación utilizada es *Sphinx* [49].

Existen varios estilos de *docstrings*, como el estilo `numpy` o el de `google`. En `scikit-rmt` se ha decidido usar el estilo seguido por Google [55], que es ampliamente usado por millones de desarrolladores en todo el mundo. La herramienta *Sphinx* incorpora ambos estilos de *docstring* con su extensión *Napoleon* [56], y se utiliza para generar la documentación final de la librería.

Finalmente, la documentación oficial del paquete se puede encontrar en *Read the Docs* [3], en el repositorio de Github [16] y en el índice de paquetes de Python PyPI tras su publicación [2].

4.3.4. Pruebas de unidad y de integración

Durante el desarrollo de un proyecto software, el código tiene que ser supervisado y probado para reducir el mayor número de errores posible.

Para ello, se implementan *tests* de unidad para cada uno de los ficheros programados en el paquete. Estas pruebas de unidad se pueden encontrar en cada módulo (`ensemble` o `covariance`) dentro del directorio `tests`. Se usa la librería de Python `pytest` [57] para ejecutar dichas pruebas y, adicionalmente, la librería `pytest-cov` [58] para controlar el flujo de los *tests* o *coverage*, es decir, para contabilizar qué líneas de código han sido ejecutadas y cuáles no. En total, las pruebas de unidad superan el 93 % de cobertura del código escrito, y sólo aquellas funciones o métodos que tienen como resultado un gráfico de `matplotlib` [59] no son comprobados con este sistema.

Cada vez que el paquete es actualizado en GitHub, ya sean sus ficheros fuente o su documentación, varios *tests* de compilación e integración son ejecutados. *Travis CI* se encarga de comprobar que las últimas modificaciones no interfieren con las dependencias, así como ejecutar los *tests* de cobertura, actualizando los *badges* de GitHub de la figura 4.4. La herramienta *Read the Docs* hace algo similar para la documentación. Con cada actualización se compila la documentación en un servidor remoto con *Sphinx* y se publica si no se produce ningún error, actualizando el *badge* correspondiente.

Figura 4.4: *Badges* de GitHub de `scikit-rmt`

Finalmente, cierta funcionalidad gráfica y simulaciones son probadas con *Jupyter Notebooks* [60], descritos en el capítulo 5.

5

Pruebas y resultados

En este capítulo se ilustran las funcionalidades de la herramienta. Se realizan simulaciones y análisis según los protocolos descritos en [19] y [21] con el fin de contrastar los resultados obtenidos con los de estos artículos.

En primer lugar, verificamos que la simulación de las matrices aleatorias es correcta. Compararemos asimismo los histogramas de sus autovalores con los esperados de acuerdo con los resultados de la Teoría de Matrices Aleatorias. Después, vemos como la teoría detrás de la tridiagonalización de los *Ensembles* Gaussiano y de Wishart nos permite acelerar el proceso de construcción de los histogramas de dichos ensembles. Finalmente, experimentamos con varios estimadores de matrices de covarianza, comparándolos usando las métricas introducidas en 2.4.2.

5.1. Muestreo y representación espectral

Con la librería `scikit-rmt` desarrollada se puede muestrar las distintas matrices aleatorias de los *ensembles* detallados en la sección 2.2. En Python, esto se puede hacer siguiendo el código del apéndice B.5.

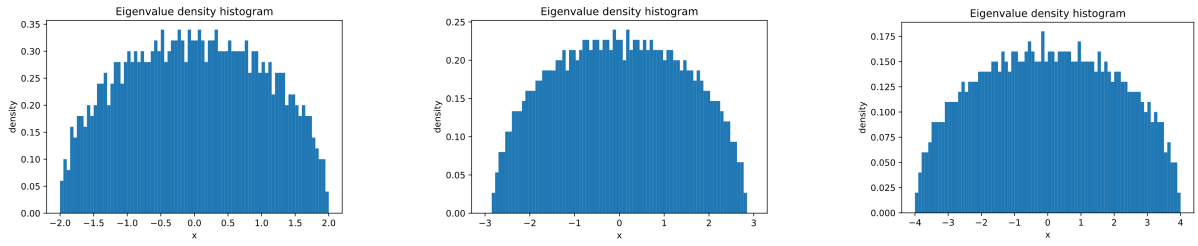
Cada vez que instanciamos un *ensemble* se muestrea una matriz aleatoria del mismo. Para generar una muestra independiente a partir de este mismo objeto se utiliza el método `goe.sample()`. En la llamada a este método se puede indicar si la matriz aleatoria a muestrar tiene que ser tridiagonal o en su forma estándar. No obstante, si queremos instanciar una matriz aleatoria tridiagonal del *Ensemble* Gaussiano o de Wishart podemos hacerlo directamente utilizando el método `goe.sample_tridiagonal()`.

No se muestran las instancias de las matrices aleatorias en concreto, ya que por inspección es difícil determinar si una matriz aleatoria sigue una cierta distribución. Las pruebas de unidad, mencionadas en la sección 4.3.4, han sido diseñadas para verificar que todas estas matrices aleatorias muestradas cumplen las propiedades básicas de cada uno de los *ensembles*, como tamaño de la matriz ($n \times n$, $p \times p$, etc.), simetría, hermiticidad...

Para detectar posibles errores en la implementación realizada para el muestreo de matrices aleatorias pertenecientes a distintos *ensembles*, se puede comparar la distribución de autovalores de las matrices generadas con la esperada de acuerdo con RMT . Para ello, basta con representar el histograma de autovalores y compararlo con la densidad espectral correspondiente a cada uno de los *ensembles*.

El método `eigval_hist(...)` calcula el histograma de autovalores, el cual es la estimación empírica de la densidad espectral de la matriz aleatoria. Este histograma se puede representar mediante el método `plot_eigval_hist(...)`.

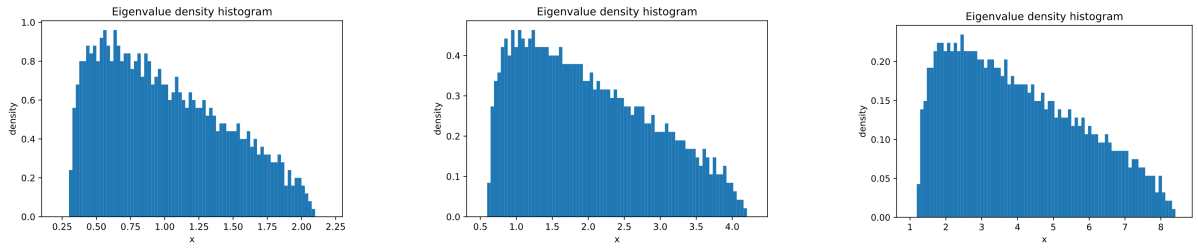
En la figura 5.1 podemos ver los histogramas espectrales de los *Ensembles* Gaussianos: GOE (histograma 5.1(a)), GUE (histograma 5.1(b)) y GSE (5.1(c)). El perfil del histograma parece aproximar el semicírculo correspondiente a la Ley de Wigner (introducida en la sección 2.3.1)que describe la densidad poblacional de los autovalores de las matrices aleatorias de Wigner, en particular, de las matrices del *Ensemble* Gaussiano.



(a) GOE 1000×1000 (b) GUE 1000×1000 (c) GSE 2000×2000

Figura 5.1: Histogramas espectrales de los tres *Ensembles* Gaussianos: GOE, GUE y GSE

La figura 5.2 ilustra las densidades espectrales de los *Ensembles* de Wishart: Wishart Real (gráfica 5.2(a)), Wishart Complejo (histograma 5.2(b)) y Wishart Cuaterniónico (5.2(c)). Su apariencia es bastante similar y, si recordamos lo explicado en la sección 2.3.1, sigue la denominada distribución de Marchenko-Pastur o Ley de Marchenko-Pastur.

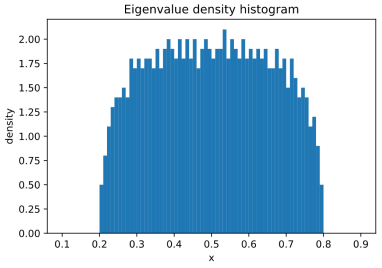


(a) WRE 1000×1000 (b) WCE 1000×1000 (c) WQE 2000×2000

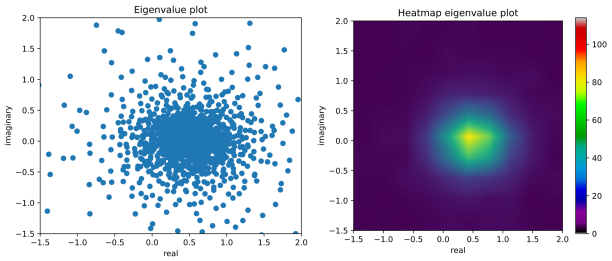
Figura 5.2: Histogramas espectrales de los tres *Ensembles* de Wishart: WRE, WCE y WQE

En relación con los *Ensembles* Manova, representados en la figura 5.3, se puede apreciar que la

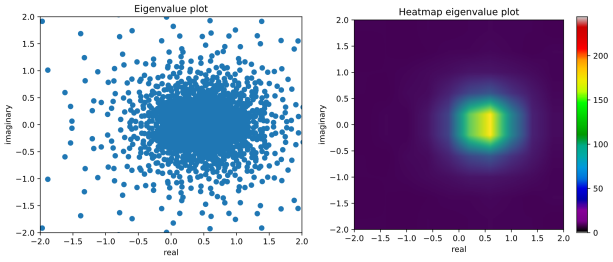
densidad espectral del *Ensemble* Real Manova 5.3(a) se localiza en el intervalo $(0{,}2, 0{,}8)$. Por otro lado, debido a que las matrices aleatorias del *Ensemble* Manova no son simétricas ni hermíticas, sus autovalores no tienen por qué ser reales. Es el caso del *Ensemble* Complejo Manova (gráfica 5.3(b)) y del *Ensemble* Cuaterniónico Manova (5.3(c)), cuyos autovalores toman valores complejos. Por esa razón se han implementado sus histogramas de tal forma que representen los autovalores individualmente, así como un mapa de calor que codifica un histograma en dos dimensiones, para poder apreciar las acumulaciones de autovalores complejos. Los colores cálidos (rojo, naranja) del mapa de calor indica una mayor densidad de autovalores, mientras que los colores oscuros indican que apenas hay muestras localizadas ahí. La densidad del MCE se acumula en el disco centrado en el 0 complejo y de radio 2, existiendo varios puntos atípicos (*outliers*) que no parecen tener ninguna característica común. En cuanto al MQE, la densidad espectral se concentra en el disco de centro el 0 complejo y de radio 5, con algunos *outliers*. Podemos ver en la gráfica 5.3(c) que los autovalores del MQE van en parejas, ya que cada autovalor va acompañado de su conjugado (fijarse en la simetría con respecto al eje real).



(a) MRE 1000×1000



(b) MCE 1000×1000

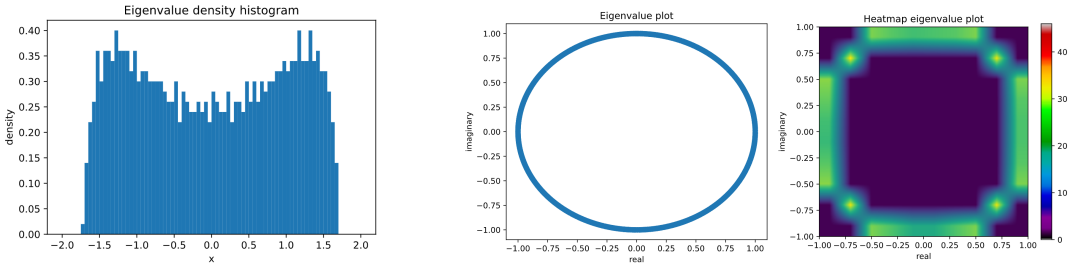


(c) MQE 2000×2000

Figura 5.3: Histogramas espectrales de los tres *Ensembles* Manova: MRE, MCE y MQE

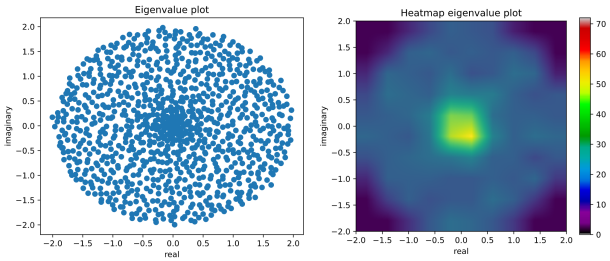
Finalmente, la figura 5.4 representa la densidad de autovalores de las matrices aleatorias de los *Ensembles* Circulares: COE (gráfica 5.4(a)), CUE (histogramas 5.4(b)) y CSE (5.4(c)).

Los autovalores del COE se encuentran en el intervalo $(-2, 2)$, concentrándose en $x = -1{,}5$ y $x = 1{,}5$. En cuanto al CUE, todos los autovalores son complejos y de módulo 1, tal y como vemos en la figura 5.4(b). Por último, los autovalores del CSE toman también valores complejos, y todos ellos se localizan en el disco centrado en 0 y de radio 2, aumentando su concentración según se acerca uno al cero complejo, tal y como se aprecia en el mapa de calor de 5.4(c).



(a) COE 1000×1000

(b) CUE 1000×1000



(c) CSE 2000×2000

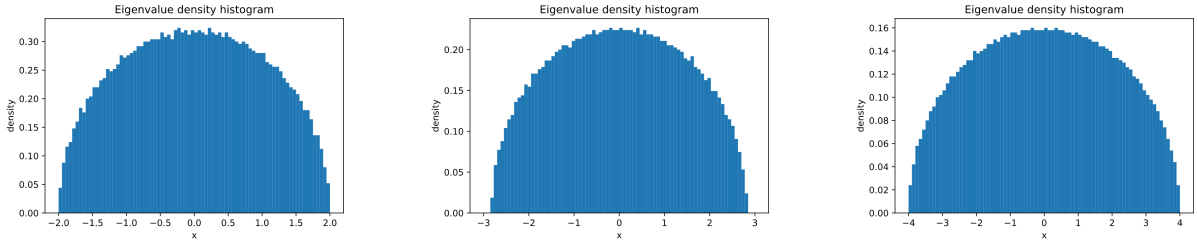
Figura 5.4: Histogramas espectrales de los tres *Ensembles* Circulares: COE, CUE y CSE

5.2. Leyes espectrales y comparación con las densidades de autovalores

Para representar los histogramas de la sección 5.1 hemos utilizado los métodos `eigval_hist(...)` y `plot_eigval_hist(...)`. No obstante, para que los histogramas acojan una apariencia cómoda para su estudio, hemos necesitado en algunos casos normalizar los autovalores por una cierta constante, así como escoger adecuadamente el intervalo de representación.

Estos detalles pueden ser muy finos para algún futuro usuario de la librería que simplemente desee visualizar y/o analizar las distintas leyes espectrales de los *ensembles* implementados. Es por ello que incorporamos ciertas funciones que permiten generar histogramas de las leyes espectrales introducidas en 2.3.1 sin tener que controlar los pequeños detalles técnicos.

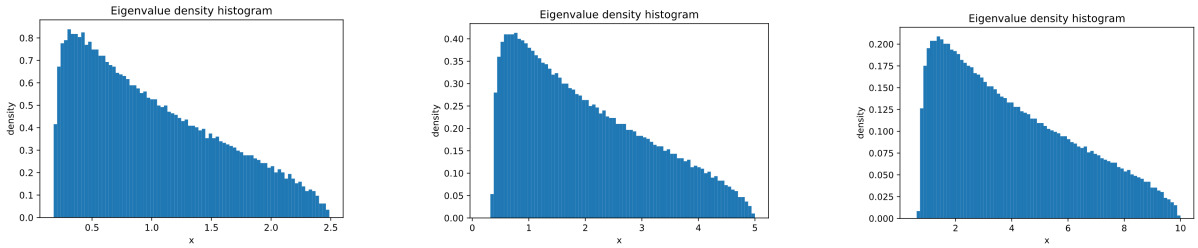
En primer lugar podemos estudiar la Ley Semicircular de Wigner, que explica el comportamiento límite de los autovalores de las matrices de Wigner, en particular, de las matrices aleatorias del *Ensemble* Gaussiano. En la figura 5.5 podemos ver el resultado de ejecutar la función implementada en el módulo `ensemble` en el fichero `plot_law.py` denominada `wigner_semicircular_law`, muestrando matrices de tamaño `n_size` = 5000 y con 80 *bins*. Para la gráfica 5.5(a) se han usado matrices GOE, para la 5.5(b) matrices GUE, y para la 5.5(c) GSE, obteniendo comportamientos muy similares.



(a) GOE 5000×5000 (b) GUE 5000×5000 (c) GSE 10000×10000

Figura 5.5: Ley Semicircular de Wigner ilustrada con los tres *Ensembles* Gaussianos: GOE, GUE y GSE

Otra ley espectral implementada es la Ley de Marchenko-Pastur, que define el comportamiento límite de los autovalores de las matrices del *Ensemble* de Wishart. En la figura 5.6 podemos ver el resultado de ejecutar la función `marchenko_pastur_law`, muestrando matrices de tamaño `n_size` = 5000 y con 80 *bins*. Para la gráfica 5.6(a) se han usado matrices del WRE, para la 5.6(b) matrices del WCE, y para la 5.6(c) del WQE, obteniendo representación muy similares.

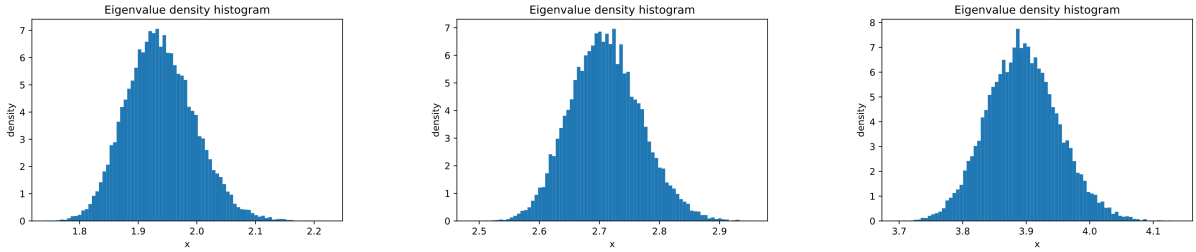


(a) GOE 5000×5000 (b) GUE 5000×5000 (c) WQE 10000×10000

Figura 5.6: Ley de Marchenko-Pastur ilustrada con los tres *Ensembles* de Wishart: WRE, WCE y WQE

Por último, permitimos estudiar la Ley de Tracy-Widom, que formula el comportamiento límite del mayor autovalor de las matrices de Wigner. Para ello se usa la función `tracy_widom_law`, muestrando matrices de tamaño `n_size` = 100. Como lógicamente cada matriz aleatoria muestrada solo tiene un mayor autovalor, necesitamos muestrar varias matrices de Wigner (en nuestro caso, del *Ensemble* Gaussiano) para poder construir el histograma de densidad del mayor autovalor. Para las simulaciones de la figura 5.7 se han muestrado 15000 matrices aleatorias para cada sub-*ensemble*. Para la gráfica 5.7(a) se han usado matrices GOE, para la 5.7(b) matrices GUE, y para la 5.7(c) GSE.

Podemos comparar cada gráfica de la figura 5.7 con las gráficas expuestas en la sección 2.3.1, viendo que se corresponden con su formulación teórica.



(a) GOE 100×100, muestrada 20000 veces (b) GUE 100×100, muestrada 20000 veces (c) GSE 100×100, muestrada 20000 veces

Figura 5.7: Ley de Tracy-Widom ilustrada con los tres *Ensembles* Gaussianos: GOE, GUE y GSE

5.3. Uso de formas tridiagonales para construcción eficiente de histogramas

Hemos analizado diferentes propiedades de los *ensembles* de matrices aleatorias a través de su densidad espectral. Para ello hemos tenido a representar gráficamente el histograma de sus autovalores, el cual es un procedimiento computacionalmente caro si seguimos el procedimiento habitual.

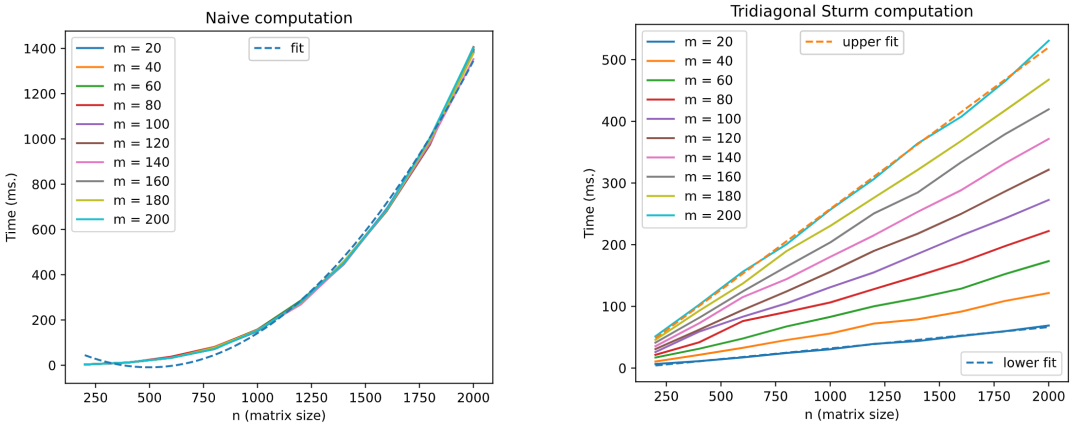
Tal y como se mencionó en 2.3.2, el algoritmo estándar para construir un histograma de autovalores consiste en fijar el intervalo de representación y el número de *bins* y calcular los valores propios de la matriz aleatoria en cuestión. Una vez calculados, se cuenta cuántos de ellos caen en cada uno de los *bins*, graficando finalmente el histograma espectral. El cálculo de autovalores mediante el algoritmo estándar DSTEQR de LAPACK [35] (el usado por la mayoría de librerías) tiene un coste computacional de $O(n^2)$ siendo n el tamaño de la matriz aleatoria, es decir, el número de autovalores. Después, hay que contar el número de autovalores por cada uno de los m *bins* fijados, lo que requiere de un tiempo extra $O(mn)$ ($O(m+n)$ si estuviesen ya ordenados al calcularlos), resultando en un coste computacional total de $O(m+n^2)$.

Sin embargo, podemos acelerar el proceso de construcción de histogramas para los *Ensembles* Gaussiano y de Wishart utilizando su forma tridiagonal (que posee exactamente los mismos autovalores) y las secuencias de Sturm, tal y como se explicó en la sección 2.3.3, obteniendo un coste computacional de $O(nm)$.

Para ratificar esto, simulamos 50 veces y promediamos el tiempo necesario para construir histogramas de los *Ensembles* Gaussiano y de Wishart usando su forma tridiagonal y las secuencias de Sturm para contabilizar los autovalores, y comparándolo con el procedimiento estándar.

La figura 5.8 contiene los resultados de las simulaciones con el *Ensemble* Gaussiano. Sin pérdida de generalidad, se muestran matrices GOE. Como podemos ver en la gráfica 5.8(a), el método estándar tiene un coste exponencial con respecto a n, y el tiempo gastado para computar su histograma es entorno 10 veces mayor que su equivalente utilizando formas tridiagonales, representado en la gráfica

5.8(b). En esta última se aprecia un comportamiento lineal con respecto a la dimensión n (número de autovalores), y que la pendiente de la recta aumenta según aumenta m (el número de *bins*), tal y como se esperaba tras el análisis del coste teórico.

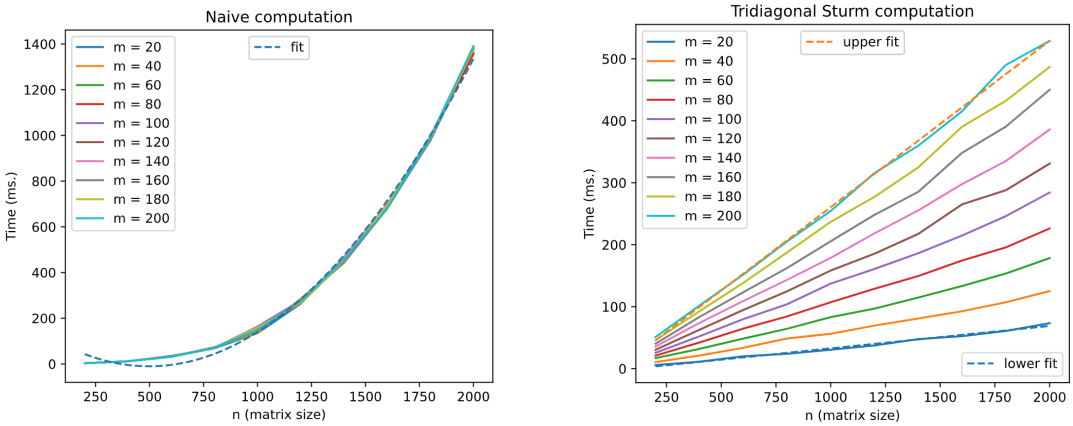


(a) Construcción de histograma estándar

(b) Construcción usando Sturm+tridiagonalización

Figura 5.8: Comparación de los tiempos de ejecución (50 repeticiones) de la construcción de histogramas del espectro del GOE mediante formas tridiagonales o mediante el algoritmo estándar

En la figura 5.9 volvemos a hacer la misma simulación con el *Ensemble* de Wishart. Sin pérdida de generalidad, se muestran matrices WRE. En la gráfica 5.9(a) vemos como el tiempo utilizado asciende exponencialmente con respecto a la variable n si usamos el método tradicional. En contraste, el método utilizando secuencias de Sturm y las formas tridiagonales de las matrices aleatorias muestra un comportamiento lineal en n y m. Esto es una mejora significativa ya que n suele ser muy grande en comparación a m cuando queremos analizar el comportamiento límite de los autovalores. En el apéndice C se puede encontrar el código para generar las simulaciones y las gráficas 5.8 y 5.9.



(a) Construcción de histograma estándar

(b) Construcción usando Sturm+tridiagonalización

Figura 5.9: Comparación de los tiempos de ejecución (50 repeticiones) de la construcción de histogramas del espectro del WRE mediante formas tridiagonales o mediante el algoritmo estándar

5.4. Estimación de matrices de covarianza

En la sección 2.4 definimos teóricamente los estimadores de matrices de covarianza a considerar para nuestra librería. Ahora, una vez implementados, vamos a analizar sus rendimientos, utilizando para ello la métrica *PRIAL* (2.12) y comparando los tiempos de ejecución.

Para poder estudiar el rendimiento de los estimadores, primero necesitamos crear una matriz de covarianzas $\mathbf{\Sigma}$ para una población, la cual será la que se intentará estimar. Influenciándonos por el estudio realizado en [21], el 20 % de los autovalores de $\mathbf{\Sigma}$ serán igual a 1, el 40 % igual a 3, y el 40 % restante igual a 10. Además, los datos seguirán una distribución normal.

Bajo las anteriores condiciones, podemos generar $\mathbf{\Sigma}$ creando una matriz diagonal $\tilde{\mathbf{I}}$ con entradas siguiendo la proporcionalidad de autovalores deseada. Después, para que $\mathbf{\Sigma}$ no sea necesariamente diagonal, será multiplicada por una matriz ortogonal $\mathbf{O}$ de la siguiente forma: $\mathbf{\Sigma} = \mathbf{O}\tilde{\mathbf{I}}\mathbf{O}^{\mathbf{T}}$.

Así pues, una vez generada la matriz de covarianzas poblacional $\mathbf{\Sigma}$, podemos muestrar los datos utilizando la función de numpy `np.random.multivariate_normal(cov=`$\mathbf{\Sigma}$`)`.

Las simulaciones realizadas serán repetidas 100 veces y sus resultados se promediarán para hallar una aproximación decente del rendimiento real de cada estimador.

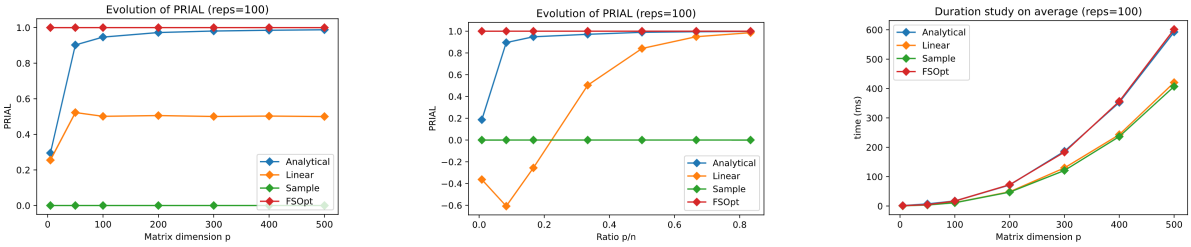
Sea p el número de atributos y, por lo tanto, el tamaño de la matriz cuadrada $\mathbf{\Sigma}$, y n el número de datos muestrados. Se estudiarán los estimadores bajo tres escenarios:

- Fijando el ratio $p/n = 1/3$, pero variando p y n. Para ello p variará en la lista [5, 50, 100, 200, 300, 400, 500] y n será el triple en cada simulación.
- Variando el ratio p/n. Para ello se mantendrá fijo el valor de $n = 600$ y p variará en la lista [5, 50, 100, 200, 300, 400, 500], provocando que p/n sea [1/120, 1/12, 1/6, 1/3, 1/2, 2/3, 5/6].
- Midiendo el coste computacional, es decir, el tiempo de ejecución, de cada estimador para $\mathbf{\Sigma}$ de tamaño p en [5, 50, 100, 200, 300, 400, 500].

En la figura 5.10 podemos ver el resultado de 100 simulaciones Monte Carlo para el estimador de covarianza muestral (*Sample* o $\mathbf{S}$), para el óptimo de muestra finita (*FSOpt* o $\mathbf{S}^*$), para el de contracción lineal (*linear*) y para el de contracción no lineal analítico (*analytical*). Tal y como se mencionó en la sección 2.4.2, los estimadores muestral y óptimo de muestra finita, por construcción, $\text{PRIAL}_n^{MV}(\mathbf{S}) = 0\,\%$ y $\text{PRIAL}_n^{MV}(\mathbf{S}^*) = 100\,\%$ respectivamente.

Podemos ver en las gráficas 5.10(a) y 5.10(b) que efectivamente $\text{PRIAL}_n^{MV}(\mathbf{S}) = 0\,\%$ y $\text{PRIAL}_n^{MV}(\mathbf{S}^*) = 100\,\%$ independientemente del valor de p y de n.

Analizando la gráfica 5.10(a), podemos ver que el estimador lineal se mantiene constante en 50 % de PRIAL a partir de $p = 50$ ($n = 150$). Para $p = 5$ y $n = 15$ tanto el estimador de contracción lineal



(a) Fijando ratio p/n (b) Variando ratio p/n (c) Coste computacional

Figura 5.10: Comparación del rendimiento, medido mediante el *PRIAL*, de los estimadores de contracción lineal y no lineal, basándose en las simulaciones de [21]

como el analítico bajan considerablemente su PRIAL, posiblemente debido a la falta de datos para estimar la matriz de covarianzas poblacional real, al ser $n = 15$ ejemplos insuficientes. No obstante, a partir de $p = 50$, $n = 150$ podemos ver que el estimador analítico de contracción no lineal supera el 90 % de PRIAL, alcanzando cerca del 100 % en $p = 500$.

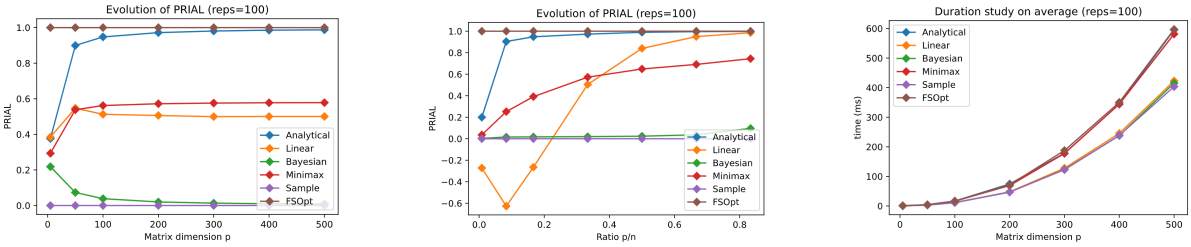
Centrándonos ahora en la gráfica 5.10(b), vemos que el estimador de contracción lineal rinde negativamente (es decir, su rendimiento es peor que el estimador muestral $\mathbf{S}$) hasta aproximadamente $p/n = 0{,}25$, donde asciende drásticamente hasta situarse muy próximo al 100 % a partir de $p/n = 0{,}8$. El PRIAL del estimador analítico de contracción no lineal converge rápidamente al 100 %, superando el 90 % a partir de $p/n = 0{,}15$.

Finalmente, analizando la gráfica de tiempos 5.10(c), podemos ver que todos los estimadores tienen un coste computacional no lineal con respecto a la dimensión p. Los estimadores muestral y de contracción lineal son los más eficientes, con unos rendimientos muy similares. Los estimadores analítico de contracción no lineal y de muestra finita FSOpt también tienen un coste muy similar, pero considerablemente mayor que los dos anteriores.

En general, por lo analizado en la figura 5.10, el estimador de contracción no lineal analítico, propuesto en [21], es el que mejor rendimiento aporta, con un PRIAL mayor y necesitando pocos datos (n pequeño o ratio p/n grande). Por ahora sería nuestro estimador de preferencia, a pesar de su coste computacional, superior al muestral y al de contracción lineal.

Estos resultados son idénticos a los obtenidos en el estudio [21], corroborando el correcto funcionamiento de los estimadores implementados en nuestra librería.

En la figura 5.11, comparamos los resultados anteriores con los de otros estimadores implementados, definidos en 2.4.1 y expuestos en [20]. Para ello, hemos vuelto a ejecutar 100 simulaciones Monte Carlo para obtener el PRIAL de cada uno de los estimadores sobre los escenarios base propuestos.



(a) Fijando ratio p/n (b) Variando ratio p/n (c) Coste computacional

Figura 5.11: Comparación del rendimiento, medido mediante el *PRIAL*, de todos los estimadores de matrices de covarianza implementados, basándose en los resultados de [20] y [21]

En primer lugar, ratificamos los resultados obtenidos en la figura 5.10 con los estimadores muestral, óptimo de muestra finita, de contracción lineal y de contracción no lineal analítico. Adicionalmente, podemos analizar los resultados de los estimadores Minimax y Empírico Bayesiano.

En la gráfica 5.11(a) vemos como el estimador Minimax supera ligeramente el rendimiento del estimador de contracción lineal, con un PRIAL constante del 55 %. Por otro lado, vemos como el estimador empírico Bayesiano empeora considerablemente según aumentan el número de atributos y datos, convergiendo a un PRIAL del 0 %, es decir, similar al estimador de covarianzas muestral.

Pasando a la gráfica 5.11(b) vemos como el estimador Minimax tiende a un 100 % de PRIAL según el ratio se acerca a 1, pero terriblemente lento. Además, el estimador empírico Bayesiano parece que rinde ligeramente mejor que el estimador muestral, pero sólo para ratios p/n cercanos a 1.

Finalmente, en la gráfica 5.11(c), vemos como el estimador Minimax tiene un coste computacional idéntico al estimador analítico y al óptimo de muestra finita, mientras que el estimador empírico bayesiano necesita de un tiempo de ejecución menor, similar al estimador muestral y al lineal.

En conclusión, el estimador Minimax es mejor que el de contracción lineal para ratios p/n pequeños (inferiores a 0.4), y el estimador empírico Bayesiano es ligeramente mejor que el muestral. No obstante, seguimos afirmando que nuestro mejor estimador es el analítico de contracción no lineal.

Para finalizar esta sección, comentar que se añaden ejemplos de uso del módulo *ensemble* en el repositorio [16], en la documentación [3] y en el apéndice D. Adicionalmente, se muestra cómo podemos reproducir las simulaciones de esta sección en el apéndice E, y también en el repositorio [16] en la documentación oficial [3].

6

Conclusión y trabajo futuro

En este trabajo hemos diseñado e implementado la librería `scikit-rmt` para el manejo y análisis de matrices aleatorias en Python. La herramienta permite muestrear matrices aleatorias pertenecientes a 12 ensembles diferentes, los cuales se encuentran entre los más utilizados y estudiados de la Random Matrix Theory: *Ensemble* Gaussiano, de Wishart, Manova y el Circular. Adicionalmente, la librería proporciona herramientas para estimar la distribución de autovalores de matrices aleatorias, visualizar la densidad de probabilidad mediante histogramas e ilustrar las leyes de Wigner, Marchenko-Pastur y Tracy-Widom, que corresponden al límite poblacional de las correspondientes distrubuciones empíricas. Además, para los *Ensembles* Gaussiano y de Wishart podemos utilizar sus formas tridiagonales para acelerar el proceso de construcción de sus histogramas espectrales. También, aplicamos la Teoría de Matrices Aleatorias para implementar estimadores de matrices de covarianza óptimos asintóticamente, basados en los estudios [21] y [20].

Se espera que la librería `scikit-rmt` proporcione herramientas útiles para matemáticos, expertos en aprendizaje automático, físicos, economistas así como investigadores y profesionales de otras áreas para el manejo de matrices aleatorias. Entre las aplicaciones de RMT se encuentran: Estadística, para generalizar el test χ^2 a varias dimensiones [10]; en Análisis Matemático, para describir errores de cálculo en operaciones con matrices [11]; en teoría de números, ya que la distribución de ceros de la función Zeta de Riemann es modelada mediante la distribución espectral de ciertas matrices aleatorias [12]; en Física, para modelar los niveles de energía de los núcleos de átomos pesados [8] y [13]; en Inteligencia Artificial, para analizar las matrices de pesos sinápticos que caracterizan el procesamiento realizado por la red y entender sus propiedades [9]; en Finanzas, para análisis de riesgos [14] o para seleccionar carteras de inversión mediante estimación de matrices de covarianza [7]; y hasta en Neurociencia, para describir la red de conexiones sinápticas entre neuronas en el cerebro [15].

Durante todo el desarrollo del trabajo hemos utilizado un gran número de herramientas y tecnologías para asegurar la creación de un producto *software* de calidad. Se ha seguido el estilo de SciKits [24] para el desarrollo de paquetes científicos de código abierto en Python aprobados y licenciados por la Open Source Initiative (OSI) [25]. Por otro lado, se usa GitHub para establecer el control de versiones, lo que también permitirá a cualquier desarrollador contribuir en la ampliación y mejora del paquete. Adicionalmente, se tienen en cuenta dos estándares de programación: la guía de estilo de

programación PEP 8 (*Style Guide for Python Code*) [28] y la guía de estilo de documentación PEP 257 (*Docstring Conventions*) [29]. La documentación se publica oficialmente en la web Read the Docs [23], ayudándose de la herramienta de generación automática de documentación *Sphinx* [49]. Para cerciorarnos de que cada modificación de la librería no originaba un error o conflicto, se ha usado la herramienta *Travis CI* [54], que también ejecutaba los *tests* de cobertura cada vez que se subía un cambio al repositorio de GitHub [16].

Por último, publicamos el *software* desarrollado en el índice de paquetes de Python PyPI en el enlace https://pypi.org/project/scikit-rmt/ [2]; y su documentación y tutoriales en la página Read the Docs en https://scikit-rmt.readthedocs.io/en/latest/ [3].

En el futuro, se podrían añadir otros *ensembles* de matrices aleatorias como los modelos de Calogero-Moser o de Ruijsenaars-Schneider, descritos en [61]. También se podría implementar la manipulación de *Formal Power series* y *Laurent series*, tomando como referencia la librería de Julia *RandomMatrices* [47], que pueden ser usadas para resolver recurrencias en Combinatoria o Teoría de Números [62]. Finalmente, podríamos aplicar los diferentes estimadores de matrices de covarianza a problemas de aprendizaje automático y analizar y mejorar los métodos de predicción (clasificación y regresión) basados en procesos gaussianos [63].

Bibliografía

[1] J. S. Marron and A. M. Alonso, "Overview of object oriented data analysis," *Biometrical Journal*, vol. 56, no. 5, pp. 732–753, 2014.

[2] A. Santorum, "scikit-rmt." PyPI distribution. https://pypi.org/project/scikit-rmt/, 2021.

[3] A. Santorum, "scikit-rmt." Read the Docs. https://scikit-rmt.readthedocs.io/en/latest/, 2021.

[4] J. Ramsay, "Functional Data Analysis." https://www.psych.mcgill.ca/misc/fda/, 2013.

[5] "Random Graphs." Lecture Notes of Electrical Engineering (UC Berkeley). https://inst.eecs.berkeley.edu/~ee126/sp18/random-graphs.pdf, 2018.

[6] V. Sazonov, "Introduction to random matrices and tensors and application to Principal Component Analysis." Lecture notes (University of Graz, Austria). https://indico.cern.ch/event/847626/contributions/3608486/attachments/1928447/3193260/presentation_Sazonov_Random_Matrices_and_Tensors.pdf, 2019.

[7] O. Ledoit and M. Wolf, "Improved estimation of the covariance matrix of stock returns with an application to portfolio selection," *Journal of Empirical Finance*, vol. 10, no. 5, pp. 603–621, 2003.

[8] E. Wigner, "Characteristic vectors of bordered matrices with infinite dimensions," *Annals of Mathematics*, vol. 62, no. 3, 1955.

[9] C. Louart, L. Zhenyu, and R. Couillet, "A random matrix approach to neural networks," *The Annals of Applied Probability*, vol. 28, no. 2, pp. 1190–1248, 2017.

[10] H. Hotelling, "The generalization of Student's Ratio," *Annals of Mathematical Statistics*, vol. 2, no. 3, pp. 360–378, 1931.

[11] J. V. Neumann and H. H. Goldstine, "Numerical inverting of matrices of high order," *Bull. Amer. Math. Soc.*, vol. 53, no. 11, pp. 1021–1099, 1947.

[12] J. Keating, "The riemann Zeta-function and Quantum Chaology," *Enrico Fermi Internat. School of Physics*, pp. 145–185, 1993.

[13] E. Wigner, "On the distribution of the roots of certain symmetric matrices," *Annals of Mathematics*, vol. 67, no. 2, 1955.

[14] O. Ledoit and M. Wolf, "Honey, I Shrunk the Sample Covariance Matrix," *The Journal of Portfolio Management*, vol. 30, no. 4, pp. 110–119, 2004.

[15] A. Almog *et al.*, "Uncovering functional signature in neural systems via random matrix theory," *PLOS Computational Biology*, vol. 15, no. 5, 2019.

[16] A. Santorum, "scikit-rmt." GitHub repository. https://github.com/AlejandroSantorum/scikit-rmt, 2021.

[17] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[18] I. Dumitriu and A. Edelman, "Matrix models for beta ensembles," *Journal of Mathematical Physics*, vol. 43, no. 11, pp. 5830–5847, 2002.

[19] J. Albrecht, C. Chan, and A. Edelman, "Sturm sequences and random eigenvalue distributions," *Foundations of Computational Mathematics*, vol. 9, no. 4, pp. 461–483, 2008.

[20] O. Ledoit and M. Wolf, "A well-conditioned estimator for large-dimensional covariance matrices," *Journal of Multivariate Analysis*, vol. 88, pp. 365–411, 2004.

[21] O. Ledoit and M. Wolf, "Analytical nonlinear shrinkage of large-dimensional covariance matrices," *Annals of Statistics*, vol. 48, no. 5, pp. 3043–3065, 2020.

[22] "Python package index." https://pypi.org/, 2021. Online; accedido el 15 de mayo de 2021.

[23] "Read the docs." https://readthedocs.org/, 2021. Online; accedido el 15 de mayo de 2021.

[24] "SciKits." https://www.scipy.org/scikits.html. Online; accedido el 15 de mayo de 2021.

[25] "Open Source Initiative." https://www.scipy.org/, 1998. Online; accedido el 15 de mayo de 2021.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[27] GAA-UAM, "scikit-fda." https://zenodo.org/record/4406983#.YIANTH0zZqM, 2020. Online; versión 0.5; accedido el 21 de abril de 2021.

[28] G. van Rossum, B. Warsaw, and N. Coghlan, "PEP 8 – Style Guide for Python Code." https://www.python.org/dev/peps/pep-0008/, 2021. Online; accedido el 17 de mayo de 2021.

[29] D. Goodger and G. van Rossum, "PEP 257 – Docstring Conventions." https://www.python.org/dev/peps/pep-0257/, 2021. Online; accedido el 17 de mayo de 2021.

[30] "Symplectic group." https://encyclopediaofmath.org/wiki/Symplectic_group, 2013. Online; accedido el 14 de mayo de 2021.

[31] "Quaternion." https://encyclopediaofmath.org/wiki/Quaternion, 2020. Online; accedido el 14 de mayo de 2021.

[32] F. Dyson, "The threefold way. Algebraic structure of symmetry groups and ensembles in quantum mechanics," *Journal of Mathematical Physics*, vol. 3, no. 6, p. 1199, 1962.

[33] C. A. Tracy and H. Widom, "On orthogonal and symplectic matrix ensembles," *Communications in Mathematical Physics*, vol. 177, no. 3, pp. 727–754, 1996.

[34] Z. D. Bai and J. W. Silverstein, *Spectral Analysis of Large Dimensional Random Matrices*. Springer, 2 ed., 2010.

[35] A. Edelman and N. R. Rao, "Random matrix theory," *Acta Numerica*, vol. 14, pp. 233–297, 2005.

[36] L. N. Trefethen and D. Bau, *Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.

[37] O. Ledoit and M. Wolf, "Nonlinear shrinkage estimation of large-dimensional covariance matricess," *Annals of Statistics*, vol. 40, no. 2, pp. 1024–1060, 2012.

[38] L. Haff, "Empirical bayes estimation of the multivariate normal covariance matrix," *Annals of Statistics*, vol. 8, pp. 586–597, 1980.

[39] D. Dey and C. Srinivasan, "Estimation of a covariance matrix under stein's loss," *Annals of Statistics*, vol. 13, no. 4, pp. 1581–1591, 1985.

[40] C. Lam, "Nonparametric eigenvalue-regularized precision or covariance matrix estimator," *Annals of Statistics*, vol. 44, pp. 928–953, 2016.

[41] O. Ledoit and M. Wolf, "Spectrum estimation: A unified framework for covariance matrix estimation and PCA in large dimensions," *Journal of Multivariate Analysis*, vol. 139, pp. 360–384, 2015.

[42] D. Berger, "empyricalRMT." https://pypi.org/project/empyricalRMT/, 2020. Librería de RMT en python; Online; versión 0.4.1; accedido el 2 de mayo de 2021.

[43] G. Giecold and L. Ouaknin, "pyRMT." https://github.com/GGiecold/pyRMT, 2017. Librería de RMT en python; Online; versión 1.0; accedido el 2 de mayo de 2021.

[44] M. Süzen, C. Weber, and J. J. Cerdà, "bristol." https://github.com/msuzen/bristol, 2019. Librería de RMT en python; Online; versión 0.2.7; accedido el 2 de mayo de 2021.

[45] B. Duthie, "RandomMatrixStabbility." https://github.com/bradduthie/RandomMatrixStability, 2020. Librería de RMT en R; Online; versión 0.1.0; accedido el 2 de mayo de 2021.

[46] A. Taqi, "RMAT." https://github.com/ataqi23/RMAT, 2021. Librería de RMT en R; Online; versión 0.2.0; accedido el 2 de mayo de 2021.

[47] J. Chen *et al.*, "RandomMatrices." https://github.com/JuliaMath/RandomMatrices.jl, 2019. Librería de RMT en Julia; Online; versión 0.5.0; accedido el 2 de mayo de 2021.

[48] D. Pavlyshyn, "RandomMatrixDistributions." https://github.com/damian-t-p/RandomMatrixDistributions.jl, 2019. Librería de RMT en Julia; Online; versión 0.3.0; accedido el 2 de mayo de 2021.

[49] G. Brandl, "Sphinx documentation." https://www.sphinx-doc.org/en/master/, 2021. Online; versión 3.5.4; accedido el 18 de marzo de 2021.

[50] "Pylint, code analysis for python." https://www.pylint.org/, 2021. Online; accedido el 15 de mayo de 2021.

[51] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser,

J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. Fernández del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, p. 357–362, 2020.

[52] R. Killip and R. Zozhan, "Matrix models and eigenvalue statistics for truncations of classical ensembles of random unitary matrices," *Communications in Mathematical Physics*, vol. 349, pp. 991–1027, 2017.

[53] M. Wolf, "Publications." https://www.econ.uzh.ch/en/people/faculty/wolf/publications.html#9, 2021. Online; accedido el 25 de marzo de 2021.

[54] "Travis CI." https://travis-ci.org/, 2021.

[55] Google, "Sphinx documentation." https://google.github.io/styleguide/pyguide.html, 2021. Online; accedido el 18 de marzo de 2021.

[56] G. Brandl and Sphinx, "Sphinx napoleon documentation." https://www.sphinx-doc.org/en/master/usage/extensions/napoleon.html, 2021. Online; versión 1.3; accedido el 18 de marzo de 2021.

[57] "pytest." https://pypi.org/project/pytest/, 2021. PyPI.

[58] "pytest-cov." PyPI distribution. https://pypi.org/project/pytest-cov/, 2021.

[59] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[60] F. Pérez and B. E. Granger, "IPython: a system for interactive scientific computing," *Computing in Science and Engineering*, vol. 9, pp. 21–29, May 2007.

[61] E. Bogomolny, O. Giraud, and C. Schmit, "Integrable random matrix ensembles," *Nonlinearity*, vol. 24, no. 11, p. 3179–3213, 2011.

[62] I. Niven, "Formal power series," *The American Mathematical Monthly*, vol. 76, no. 8, pp. 871–889, 1969.

[63] C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*. The MIT Press, 2006.

[64] R. Beezer, "A first course in linear algebra. Section SD similarity and diagonalization." http://linear.ups.edu/jsmath/0299/fcla-jsmath-2.99li49.html, 2007. Online; versión 2.99; accedido el 20 de febrero de 2021.

[65] R. Hildebrand, "Householder numerically with Mathematica." http://buzzard.ups.edu/courses/2007spring/projects/hildebrand-paper-revised.pdf, 2007. Online; Proyecto del curso 'Advanced Linear Algebra (Math 420A, Spring 2007)'.

Acrónimos

COE Circular Orthogonal Ensemble.

CSE Circular Symplectic Ensemble.

CUE Circular Unitary Ensemble.

GOE Gaussian Orthogonal Ensemble.

GSE Gaussian Symplectic Ensemble.

GUE Gaussian Unitary Ensemble.

MCE Manova Complex Ensemble.

MQE Manova Quaternion Ensemble.

MRE Manova Real Ensemble.

OSI Open Source Initiative.

RMT Random Matrix Theory.

WCE Wishart Complex Ensemble.

WQE Wishart Quaternion Ensemble.

WRE Wishart Real Ensemble.

Apéndices

A

TRIDIAGONALIZACIÓN

Aquí estudio uno de los métodos para tridiagonalizar matrices, conocido como el **método de Householder** o **reducción de Householder**.

Recordamos el concepto de matriz tridiagonal.

Definición A.0.1. *La matriz* $\mathbf{A} = (a_{ij})_{1\leq i,j\leq n} \in \mathcal{M}_n$ *es tridiagonal si* $a_{ij} = 0$ *para todo* $i \geq j+2$ *y para todo* $j \geq i+2$*, con* $i, j \in \{1, \ldots, n\}$.

A.1. Método de Householder

El método de Householder o el método de reflexión es un procedimiento utilizado para convertir matrices reales simétricas a su forma tridiagonal.

En el proceso se obtiene la descomposición $\mathbf{A} = \mathbf{PTP}^T$, donde $\mathbf{P}$ es una matriz ortogonal y $\mathbf{T}$ es una matriz simétrica tridiagonal. Notar que $\mathbf{A}$ y $\mathbf{T}$ son matrices similares gracias a que podemos escribir $\mathbf{A}$ como $\mathbf{PTP}^T$ (definición `SIM` de [64]). Esta propiedad será útil en la próxima subsección.

El proceso de tridiagonalización mediante el método de Householder es un procedimiento iterativo donde en cada iteración se tridiagonaliza una columna y fila, empezando en la primera fila y columna, siguiendo con la segunda fila y columna y continuando hasta la fila y columna n-ésima.

Es importante observar que la matriz $\mathbf{P}$ es formada a partir del producto de matrices de reflexión, donde cada una de ellas está determinada por un cierto vector. Estas matrices de reflexión se encargan, en cada iteración del método, de tridiagonalizar una fila y columna de la matriz original, es decir, anular todas las entradas de la fila y columna en cuestión, menos los elementos de la diagonal, de la super-diagonal y de la sub-diagonal.

El algoritmo A.1 describe el método de Householder, expuesto con detalle en [65].

Con este método podemos tridiagonalizar las matrices aleatorias de los *Ensembles* Gaussianos y de Wishart (entre otros). Podríamos utilizar su forma tridiagonal para calcular su espectro de autovalores de forma más eficiente, no obstante hay que resaltar que el propio proceso de tridiagonalización

input : matriz $\mathbf{A}$ real simétrica de tamaño $n \times n$
output: matriz $\mathbf{T}$ tridiagonal simétrica $n \times n$ y, opcionalmente, matriz $\mathbf{P}$ ortogonal $n \times n$

A_list $\leftarrow [A]$;
H_list $\leftarrow []$;
for $j \leftarrow 0$ **to** $n-3$ **do**
  if $A[j+1,j] \geq 0$ **then** $\alpha = -\sqrt{\sum_{k=j+1}^{n} A[k,j]^2}$;
  else $\alpha = \sqrt{\sum_{k=j+1}^{n} A[k,j]^2}$;
  $r = \sqrt{\frac{\alpha^2}{2} - \frac{\alpha}{2} A[j+1,j]}$;
  $x =$ Zeros$(n,1)$; \\vector columna de ceros
  $x[j+1] = \frac{A[j+1,j]-\alpha}{2r}$;
  for $k \leftarrow j+2$ **to** $n-1$ **do**
    $x[k] = \frac{A[k,j]}{2r}$;
  end
  $H = I_n - 2xx^T$;
  $A = HAH$;
  H_list = Append(H);
  A_list = Append(A);
end
$\mathbf{T} \leftarrow A$;
$\mathbf{P} \leftarrow$ H_list[0] $\cdot$ H_list[1] $\cdot \ldots \cdot$ H_list[$n-2$];

Algoritmo A.1: Algoritmo del método de Householder

consume parte del tiempo ahorrado en la construcción del histograma, ya que el método de Householder tiene un coste computacional de $O(n^2)$, donde n es el tamaño de la matriz simétrica cuadrada a tridiagonalizar. Por lo tanto, la potencia de la forma tridiagonal de dichas matrices aleatorias no reside en el hecho de que se puedan tridiagonalizar, sino en que sus formas tridiagonales poseen los mismos autovalores, tal y como indicamos en 2.3.3, las cuales las podemos muestrar directamente.

B

Muestreo de *Ensembles* de matrices aleatorias

En este capítulo exponemos cómo podríamos muestrar las matrices aleatorias de los *ensembles* considerados. Es importante destacar el uso de la librería de cálculo científico `numpy` para acelerar el proceso de computación, ya que ciertos cálculos sobre matrices tienden a ser temporalmente costosos.

El código está basado en los artículos [18] y [52].

```python
import numpy as np

# Sampling Gaussian Orthogonal Ensemble
def sample_goe(self):
    # n by n matrix of random Gaussians
    mtx = np.random.randn(self.n,self.n)
    # symmetrize matrix
    self.matrix = (mtx + mtx.transpose())/2
    return self.matrix

# Sampling Gaussian Unitary Ensemble
def sample_gue(self):
    # n by n random complex matrix
    mtx = np.random.randn(self.n,self.n) + (0+1j)*np.random.randn(self.n,self.n)
    # hermitian matrix
    self.matrix = (mtx + mtx.transpose())/2
    return self.matrix

# Sampling Gaussian Symplectic Ensemble
def sample_gse(self):
    # n by n random complex matrix
    x_mtx = np.random.randn(self.n,self.n) + (0+1j)*np.random.randn(self.n,self.n)
    # another n by n random complex matrix
    y_mtx = np.random.randn(self.n,self.n) + (0+1j)*np.random.randn(self.n,self.n)
    # [X Y; -conj(Y) conj(X)]
    mtx = np.block([
                    [x_mtx               , y_mtx],
                    [-np.conjugate(y_mtx), np.conjugate(x_mtx)]
                     ])
    # hermitian matrix
    self.matrix = (mtx + mtx.transpose())/2
    return self.matrix
```

Código B.1: Muestreo de matrices aleatorias del *Ensemble* Gaussiano en Python

```python
import numpy as np

# Sampling Wishart Real Ensemble
def sample_wre(self):
    # p by n matrix of random Gaussians
    mtx = np.random.randn(self.p,self.n)
    # symmetrize matrix
    self.matrix = np.matmul(mtx, mtx.transpose())
    return self.matrix

# Sampling Wishart Complex Ensemble
def sample_wce(self):
    # p by n random complex matrix of random Gaussians
    mtx = np.random.randn(self.p,self.n) + (0+1j)*np.random.randn(self.p,self.n)
    # hermitian matrix
    self.matrix = np.matmul(mtx, mtx.transpose())
    return self.matrix

# Sampling Wishart Quaternion Ensemble
def sample_wqe(self):
    # p by n random complex matrix of random Gaussians
    x_mtx = np.random.randn(self.p,self.n) + (0+1j)*np.random.randn(self.p,self.n)
    # p by n random complex matrix of random Gaussians
    y_mtx = np.random.randn(self.p,self.n) + (0+1j)*np.random.randn(self.p,self.n)
    # [X Y; -conj(Y) conj(X)]
    mtx = np.block([
                    [x_mtx              , y_mtx],
                    [-np.conjugate(y_mtx), np.conjugate(x_mtx)]
                ])
    # hermitian matrix
    self.matrix = np.matmul(mtx, mtx.transpose())
    return self.matrix
```

Código B.2: Muestreo de matrices aleatorias del *Ensemble* de Wishart en Python

```python
import numpy as np

# Sampling Manova Real Ensemble
def sample_mre(self):
    # m by n1 random real matrix of random Gaussians
    x_mtx = np.random.randn(self.m,self.n1)
    # m by n2 random real matrix of random Gaussians
    y_mtx = np.random.randn(self.m,self.n2)
    # A1 = X * X'
    a1_mtx = np.matmul(x_mtx, x_mtx.transpose())
    # A2 = X * X' + Y * Y'
    a2_mtx = a1_mtx + np.matmul(y_mtx, y_mtx.transpose())
    # A = (X * X') / (X * X' + Y * Y') = (X * X') * (X * X' + Y * Y')^(-1)
    self.matrix = np.matmul(a1_mtx, np.linalg.inv(a2_mtx))
    return self.matrix

# Sampling Manova Complex Ensemble
def sample_mce(self):
    # m by n1 random complex matrix of random Gaussians
    x_mtx = np.random.randn(self.m,self.n1) + (0+1j)*np.random.randn(self.m,self.n1)
    # m by n2 random complex matrix of random Gaussians
```

```
    y_mtx = np.random.randn(self.m,self.n2) + (0+1j)*np.random.randn(self.m,self.n2)
    # A1 = X * X'
    a1_mtx = np.matmul(x_mtx, x_mtx.transpose())
    # A2 = X * X' + Y * Y'
    a2_mtx = a1_mtx + np.matmul(y_mtx, y_mtx.transpose())
    # A = (X * X') / (X * X' + Y * Y') = (X * X') * (X * X' + Y * Y')^(-1)
    self.matrix = np.matmul(a1_mtx, np.linalg.inv(a2_mtx))
    return self.matrix

# Sampling Manova Quaternion Ensemble
def sample_mqe(self):
    # m by n1 random complex matrix of random Gaussians
    x1_mtx = np.random.randn(self.m,self.n1) + (0+1j)*np.random.randn(self.m,self.n1)
    # m by n1 random complex matrix of random Gaussians
    x2_mtx = np.random.randn(self.m,self.n1) + (0+1j)*np.random.randn(self.m,self.n1)
    # m by n2 random complex matrix of random Gaussians
    y1_mtx = np.random.randn(self.m,self.n2) + (0+1j)*np.random.randn(self.m,self.n2)
    # m by n2 random complex matrix of random Gaussians
    y2_mtx = np.random.randn(self.m,self.n2) + (0+1j)*np.random.randn(self.m,self.n2)
    # X = [X1 X2; -conj(X2) conj(X1)]
    x_mtx = np.block([
                    [x1_mtx               , x2_mtx],
                    [-np.conjugate(x2_mtx), np.conjugate(x1_mtx)]
                    ])
    # Y = [Y1 Y2; -conj(Y2) conj(Y1)]
    y_mtx = np.block([
                     [y1_mtx               , y2_mtx],
                     [-np.conjugate(y2_mtx), np.conjugate(y1_mtx)]
                    ])
    # A1 = X * X'
    a1_mtx = np.matmul(x_mtx, x_mtx.transpose())
    # A2 = X * X' + Y * Y'
    a2_mtx = a1_mtx + np.matmul(y_mtx, y_mtx.transpose())
    # A = (X * X') / (X * X' + Y * Y') = (X * X') * (X * X' + Y * Y')^(-1)
    self.matrix = np.matmul(a1_mtx, np.linalg.inv(a2_mtx))
    return self.matrix
```

Código B.3: Muestreo de matrices aleatorias del *Ensemble* Manova en Python

```
import numpy as np

# Sampling Circular Orthogonal Ensemble
def sample_coe(self):
    # sampling unitary Haar-distributed matrix
    u_mtx = _sample_haar_mtx(self.n)
    # mapping to Circular Orthogonal Ensemble
    self.matrix = np.matmul(u_mtx.transpose(), u_mtx)
    return self.matrix

# Sampling Circular Unitary Ensemble
def sample_cue(self):
    # sampling unitary Haar-distributed matrix
    self.matrix = _sample_haar_mtx(self.n)
    return self.matrix

# Sampling Circular Symplectic Ensemble
```

```
def sample_cse(self):
    # sampling unitary Haar-distributed matrix of size 2n
    u_mtx = _sample_haar_mtx(2*self.n)
    # mapping to Circular Symplectic Ensemble
    j_mtx = self._build_j_mtx()
    # U_R = J * U^T * J^T
    u_r_aux = np.matmul(j_mtx, u_mtx.transpose())
    u_r_mtx = np.matmul(u_r_aux, j_mtx.transpose())
    # A = U^R * U
    self.matrix = np.matmul(u_r_mtx, u_mtx)
    return self.matrix
```

Código B.4: Muestreo de matrices aleatorias del *Ensemble* Circular en Python

Estos métodos son implementados por nuestra librería, encapsulándolos en el método abstracto `sample`. El uso de un tipo de método de muestro u otro dependerá del parámetro `beta` de cada *ensemble*. Como regla general, `beta` = 1 indica que la matriz aleatoria tendrá entrada en el cuerpo de los reales, `beta` = 2 en los complejos, y `beta` = 4 en el cuerpo cuaterniónico.

En el código modelo B.5 podemos ver un ejemplo de uso de nuestra interfaz para generar instancias de matrices aleatorias de los diferentes *ensembles* implementados.

```
# importing ensembles
from skrmt.ensemble import GaussianEnsemble
from skrmt.ensemble import WishartEnsemble
from skrmt.ensemble import ManovaEnsemble
from skrmt.ensemble import CircularEnsemble

"""
    Some examples of how to sample random matrices of Gaussian Ensemble
"""
# sampling GOE (standard form) of size 100x100
goe = GaussianEnsemble(beta=1, n=100)
# get sampled matrix to do something
mtx = goe.matrix
# resampling
mtx = goe.sample()
# sampling GOE (tridiagonal form) of size 100x100
goe = GaussianEnsemble(beta=1, n=100, use_tridiagonal=True)
# resampling in its tridiagonal form (use_tridiagonal attribute is True)
mtx = goe.sample()
# or
mtx = goe.sample_tridiagonal()

# sampling GUE (standard form) of size 100x100
gue = GaussianEnsemble(beta=2, n=100)
# sampling GUE (tridiagonal form) of size 100x100
gue = GaussianEnsemble(beta=2, n=100, use_tridiagonal=True)

# sampling GSE (standard form) of size 100x100
gse = GaussianEnsemble(beta=4, n=50)
# sampling GSE (tridiagonal form) of size 100x100
gse = GaussianEnsemble(beta=4, n=50, use_tridiagonal=True)

```

```
33
34 """
35     Some examples of how to sample random matrices of Wishart Ensemble
36 """
37 # sampling WRE (standard form) of size 100x100
38 wre = WishartEnsemble(beta=1, p=100, n=1000)
39 # sampling WRE (tridiagonal form) of size 100x100
40 wre = WishartEnsemble(beta=1, p=100, n=1000, use_tridiagonal=True)
41
42 # sampling WCE (standard form) of size 100x100
43 wce = WishartEnsemble(beta=2, p=100, n=1000)
44 # sampling WCE (tridiagonal form) of size 100x100
45 wce = WishartEnsemble(beta=2, p=100, n=1000, use_tridiagonal=True)
46
47 # sampling WQE (standard form) of size 100x100
48 wqe = WishartEnsemble(beta=4, p=50, n=500)
49 # sampling WQE (tridiagonal form) of size 100x100
50 wqe = WishartEnsemble(beta=4, p=50, n=500, use_tridiagonal=True)
51
52
53 """
54     Some examples of how to sample random matrices of Manova Ensemble
55 """
56 # sampling MRE of size 100x100
57 mre = ManovaEnsemble(beta=1, m=100, n1=1000, n2=1000)
58 # sampling MCE of size 100x100
59 mce = ManovaEnsemble(beta=2, m=100, n1=1000, n2=1000)
60 # sampling MQE of size 100x100
61 mqe = ManovaEnsemble(beta=4, m=50, n1=500, n2=500)
62
63
64 """
65     Some examples of how to sample random matrices of Circular Ensemble
66 """
67 # sampling COE of size 100x100
68 coe = CircularEnsemble(beta=1, n=100)
69 # sampling CUE of size 100x100
70 cue = CircularEnsemble(beta=2, n=100)
71 # sampling CSE of size 100x100
72 cse = CircularEnsemble(beta=4, n=50)
```

Código B.5: Interfaz de muestreo de las matrices aleatorias de los *ensembles* implementados

C

Construcción eficiente de histogramas

En la porción de código C.1 se muestran las instrucciones para realizar las simulaciones que comparan el rendimiento de la construcción de histogramas de los *Ensembles* Gaussiano y de Wishart mediante su forma tridiagonal y la teoría de las secuencias de Sturm (recordar sección 2.3.3), o utilizando el algoritmo estándar DSTEQR de LAPACK.

```
import time
import numpy as np
import matplotlib.pyplot as matplt

from skrmt.ensemble import GaussianEnsemble
from skrmt.ensemble import WishartEnsemble

# Auxiliary function
def _build_m_labels(M):
    labels = [("m = "+str(m)) for m in M]
    return labels

# Common function to be used by the following two
def _plot_times(N_list, bins_list, times_naive, times_tridiag):
    # creating subplots
    fig, axes = plt.subplots(nrows=1, ncols=2)
    fig.set_figheight(5)
    fig.set_figwidth(13)
    fig.subplots_adjust(hspace=.5)

    # labels for plots and nodes for interpolation
    labels = _build_m_labels(bins_list)
    nodes = np.linspace(N_list[0], N_list[-1], 1000)

    # Fitting line naive computation
    y_naive = times_naive[:,0]
    a, m, b = np.polyfit(N_list, y_naive, 2)
    y_smooth_naive = a*nodes**2 + m*nodes + b
    #spl = make_interp_spline(N_list, y_naive, k=2)
    #y_smooth_naive = spl(nodes)

    # Fitting line tridiagonal computational (smallest bin)
    y_tridiag_low = times_tridiag[:,0]
    m, b = np.polyfit(N_list, y_tridiag_low, 1)
    y_smooth_tridiag_low = m*nodes + b
    #spl = make_interp_spline(N_list, y_tridiag_low, k=1)
```

```
    #y_smooth_tridiag_low = spl(nodes)

    # Fitting line tridiagonal computational (largest bin)
    y_tridiag_up = times_tridiag[:,-1]
    m, b = np.polyfit(N_list, y_tridiag_up, 1)
    y_smooth_tridiag_up = m*nodes + b
    #spl = make_interp_spline(N_list, y_tridiag_up, k=1)
    #y_smooth_tridiag_up = spl(nodes)

    # Naive plot
    lines = axes[0].plot(N_list, times_naive)
    fit_line = axes[0].plot(nodes, y_smooth_naive, '--')
    legend1 = axes[0].legend(lines, labels, loc=0)
    legend2 = axes[0].legend(fit_line, ['fit'], loc=9)
    axes[0].add_artist(legend1)
    axes[0].add_artist(legend2)
    axes[0].set_title('Naive computation')
    axes[0].set_xlabel('n (matrix size)')
    axes[0].set_ylabel('Time (ms.)')

    # Tridiagonal plot
    lines = axes[1].plot(N_list, times_tridiag)
    fit_line1 = axes[1].plot(nodes, y_smooth_tridiag_low, '--')
    fit_line2 = axes[1].plot(nodes, y_smooth_tridiag_up, '--')
    legend1 = axes[1].legend(lines, labels, loc=0)
    legend2 = axes[1].legend(fit_line1, ['lower fit'], loc=4)
    legend3 = axes[1].legend(fit_line2, ['upper fit'], loc=9)
    axes[1].add_artist(legend1)
    axes[1].add_artist(legend2)
    axes[1].add_artist(legend3)
    axes[1].set_title('Tridiagonal Sturm computation')
    axes[1].set_xlabel('n (matrix size)')
    _ = axes[1].set_ylabel('Time (ms.)')

    plt.show()


# Creating graphic using GOE (tridiagonal vs standard)
def gaussian_tridiagonal_sim(N_list, bins_list, nreps=10):
    # time lists
    times_naive = np.zeros((len(N_list), len(bins_list)))
    times_tridiag = np.zeros((len(N_list), len(bins_list)))

    # default interval and norm const
    interval = (-2, 2)
    to_norm = False

    # simulating times
    for (i, n) in enumerate(N_list):
        for (j, m) in enumerate(bins_list):
            for _ in range(nreps):
                goe1 = GaussianEnsemble(beta=1, n=n, use_tridiagonal=False)
                t1 = time.time()
                eig_hist_nt, bins_nt = goe1.eigval_hist(bins=m, interval=interval, density=to_norm
    )
                t2 = time.time()
```

```
                times_naive[i][j] += (t2 - t1)*1000 # ms

                goe2 = GaussianEnsemble(beta=1, n=n, use_tridiagonal=True)
                t1 = time.time()
                eig_hist_nt, bins_nt = goe2.eigval_hist(bins=m, interval=interval, density=to_norm
    )
                t2 = time.time()
                times_tridiag[i][j] += (t2 - t1)*1000 # ms

            times_naive[i][j] /= nreps
            times_tridiag[i][j] /= nreps

    _plot_times(N_list, bins_list, times_naive, times_tridiag)


# Creating graphic using WRE (tridiagonal vs standard)
def wishart_tridiagonal_sim(N_list, bins_list, nreps=10):
    # time lists
    times_naive = np.zeros((len(N_list), len(bins_list)))
    times_tridiag = np.zeros((len(N_list), len(bins_list)))

    # default interval and norm const
    interval = (0, 4)
    to_norm = False

    # simulating times
    for (i, n) in enumerate(N_list):
        for (j, m) in enumerate(bins_list):
            for _ in range(nreps):
                wre1 = WishartEnsemble(beta=1, p=n, n=3*n, use_tridiagonal=False)
                t1 = time.time()
                eig_hist_nt, bins_nt = wre1.eigval_hist(bins=m, interval=interval, density=to_norm
    )
                t2 = time.time()
                times_naive[i][j] += (t2 - t1)*1000 # ms

                wre2 = WishartEnsemble(beta=1, p=n, n=3*n, use_tridiagonal=True)
                t1 = time.time()
                eig_hist_nt, bins_nt = wre2.eigval_hist(bins=m, interval=interval, density=to_norm
    )
                t2 = time.time()
                times_tridiag[i][j] += (t2 - t1)*1000 # ms

            times_naive[i][j] /= nreps
            times_tridiag[i][j] /= nreps

    _plot_times(N_list, bins_list, times_naive, times_tridiag)

# main
if __name__ == '__main__':
    # matrix sizes
    n_list = [200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000]
    # number of bins
    m_list = [20, 40, 60, 80, 100, 120, 140, 160, 180, 200]

    # GOE hist (tridiagonal vs standard)
```

```
145     gaussian_tridiagonal_sim(N_list=n_list, bins_list=m_list, nreps=50)
146
147     # WRE hist (tridiagonal vs standard)
148     wishart_tridiagonal_sim(N_list=n_list, bins_list=m_list, nreps=50)
```

Código C.1: Simulaciones para comparar los tiempos de construcción de histogramas mediante formas tridiagonales o mediante el algoritmo estándar de LAPACK

D

Ejemplos de uso módulo *ensemble*

En este apéndice se ilustra brevemente cómo podemos utilizar el paquete `scikit-rmt` desarrollado para el muestreo, análisis y simulación de los *ensembles* de matrices aleatorias considerados.

Use examples of *ensemble* module

In [1]:

```
%load_ext autoreload
%autoreload 2
```

In [76]:

```
import time
import numpy as np
import matplotlib.pyplot as plt

from skrmt.ensemble import GaussianEnsemble
from skrmt.ensemble import WishartEnsemble
from skrmt.ensemble import ManovaEnsemble
from skrmt.ensemble import CircularEnsemble
from skrmt.ensemble import wigner_semicircular_law
from skrmt.ensemble import marchenko_pastur_law
from skrmt.ensemble import tracy_widom_law
```

1. Sampling

In this section we sample all the implemented random matrices. A human cannot discern their distribution, but it is an example on how to use the library interface.

1.1 Gaussian Ensemble

In [4]:

```
goe = GaussianEnsemble(beta=1, n=3)
print('GOE:\n', goe.matrix)
gue = GaussianEnsemble(beta=2, n=3)
print('GUE:\n', gue.matrix)
gse = GaussianEnsemble(beta=4, n=2)
print('GSE:\n', gse.matrix)
```

```
GOE:
 [[ 0.34574696 -0.10802385  0.38245343]
 [-0.10802385 -0.60113963  0.28624612]
 [ 0.38245343  0.28624612 -0.96503739]]
GUE:
 [[-1.75776052+1.34059319j -1.03653541-0.55697515j -0.33244793-0.8
3864516j]
 [-1.03653541-0.55697515j  0.96619151-0.09889716j -0.76077026+0.00
288841j]
 [-0.33244793-0.83864516j -0.76077026+0.00288841j  1.13704881-1.92
652204j]]
GSE:
 [[-1.00518576-0.23403217j -0.76288162-0.59919478j  0.        -0.7
0560753j
   0.71712613-0.08608688j]
 [-0.76288162-0.59919478j -0.00194212-1.1177993j  -0.71712613-0.08
608688j
   0.        -0.92899929j]
 [ 0.        -0.70560753j -0.71712613-0.08608688j -1.00518576+0.23
403217j
  -0.76288162+0.59919478j]
 [ 0.71712613-0.08608688j  0.        -0.92899929j -0.76288162+0.59
919478j
  -0.00194212+1.1177993j ]]
```

1.2 Wishart Ensemble

In [6]:

```
wre = WishartEnsemble(beta=1, p=3, n=5)
print('WRE:\n', wre.matrix)
wce = WishartEnsemble(beta=2, p=3, n=5)
print('WCE:\n', wce.matrix)
wqe = WishartEnsemble(beta=4, p=2, n=4)
print('WQE:\n', wqe.matrix)
```

```
WRE:
 [[ 2.46931112 -0.57386042  1.77196454]
 [-0.57386042  3.78201121 -0.22010238]
 [ 1.77196454 -0.22010238  2.58638309]]
WCE:
 [[-1.24134873+6.68369789j  2.02830625-2.30522714j  2.7145336 -8.8
0343238j]
 [ 2.02830625-2.30522714j  0.05889768-2.81056755j  0.30486331+0.76
986885j]
 [ 2.7145336 -8.80343238j  0.30486331+0.76986885j  8.09438224+7.46
45416j ]]
WQE:
 [[-1.19791120e+01 +6.93072529j  3.53799596e+00 -7.89734075j
  -1.01894490e-15 -5.69965262j -6.80379062e+00 -5.32326846j]
 [ 3.53799596e+00 -7.89734075j  1.47842964e+00 -6.66690797j
    6.80379062e+00 -5.32326846j  1.70848363e-16-12.82831687j]
 [-1.01894490e-15 -5.69965262j  6.80379062e+00 -5.32326846j
  -1.19791120e+01 -6.93072529j  3.53799596e+00 +7.89734075j]
 [-6.80379062e+00 -5.32326846j  1.70848363e-16-12.82831687j
    3.53799596e+00 +7.89734075j  1.47842964e+00 +6.66690797j]]
```

1.3 Manova Ensemble

In [9]:

```
mre = ManovaEnsemble(beta=1, m=3, n1=5, n2=5)
print('MRE:\n', mre.matrix)
mce = ManovaEnsemble(beta=2, m=3, n1=5, n2=5)
print('MCE:\n', mce.matrix)
mqe = ManovaEnsemble(beta=4, m=2, n1=4, n2=4)
print('MQE:\n', mqe.matrix)
```

```
MRE:
 [[ 0.82903551  0.2813032  -0.51276461]
 [ 0.00978011  0.37836247  0.18055985]
 [-0.21244069  0.11478935  0.28445309]]
MCE:
 [[ 0.44949186+0.35241057j  0.36975611+0.11653222j  0.41337193+1.3
1171215j]
 [-0.01077454+0.25636807j  0.57258515+0.49158639j -0.82024918+0.86
580046j]
 [-0.07154412-0.08621934j -0.1976965 -0.20859239j  0.95197131-0.65
757957j]]
MQE:
 [[ 0.4732006 +0.31192239j  0.31537065+0.31799136j -0.32324387-0.5
9939031j
  -0.38846748+0.04045222j]
 [-0.4027095 -0.30799041j  0.3561037 -0.20572875j  0.1496378 +0.79
980265j
  -0.04497396-0.08495502j]
 [ 0.32324387-0.59939031j  0.38846748+0.04045222j  0.4732006 -0.31
192239j
   0.31537065-0.31799136j]
 [-0.1496378 +0.79980265j  0.04497396-0.08495502j -0.4027095 +0.30
799041j
   0.3561037 +0.20572875j]]
```

1.4 Circular Ensemble

In [11]:

```
coe = CircularEnsemble(beta=1, n=3)
print('COE:\n', coe.matrix)
cue = CircularEnsemble(beta=2, n=3)
print('CUE:\n', cue.matrix)
cse = CircularEnsemble(beta=4, n=2)
print('CSE:\n', cse.matrix)
```

```
COE:
 [[ 0.77487309-0.13495178j  0.10114703+0.19286609j  0.55935399-0.1
4510264j]
 [ 0.10114703+0.19286609j  0.29263288+0.77675739j -0.46869198-0.20
955605j]
 [ 0.55935399-0.14510264j -0.46869198-0.20955605j -0.62572154+0.10
466601j]]
CUE:
 [[-0.23057877-0.07197403j -0.58029703+0.51171069j -0.47169537+0.3
4722352j]
 [ 0.21500119+0.35597382j -0.29054845+0.50393713j  0.27008043-0.64
478108j]
 [-0.83615037+0.26374772j  0.25050759-0.01680561j -0.21504723-0.34
929631j]]
CSE:
 [[ 2.06388227e-01-5.42398419e-02j -7.37507562e-19-2.77555756e-17j
  -2.40127224e-01-9.27359309e-01j  4.74243258e-01+9.01129856e-02j]
 [-4.13797022e-01-2.89745858e-01j  4.46515450e-01+8.73119467e-01j
  -4.13797022e-01-2.89745858e-01j -8.63427834e-01-1.45691543e+00j]
 [-8.57869530e-01-7.68947520e-01j -4.74243258e-01-9.01129856e-02j
    4.52073755e-01+1.56108737e+00j -4.74243258e-01-9.01129856e-02j]
 [ 4.13797022e-01+2.89745858e-01j -2.40127224e-01-9.27359309e-01j
  -2.10096004e-17+4.16333634e-17j  2.11946531e-01+6.33728064e-01
j]]
```

2. Histogramming spectral density

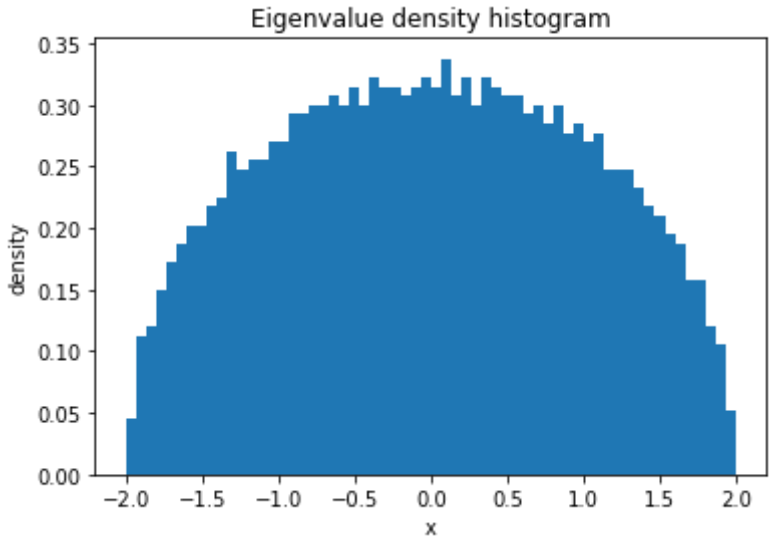
Spectral density histogram is shown. Note that the representation interval is important to visualize correctly the spectral distribution.

2.1 Gaussian Ensemble

Gaussian Ensemble eigenvalues follow Wigner's Semicircle Law.

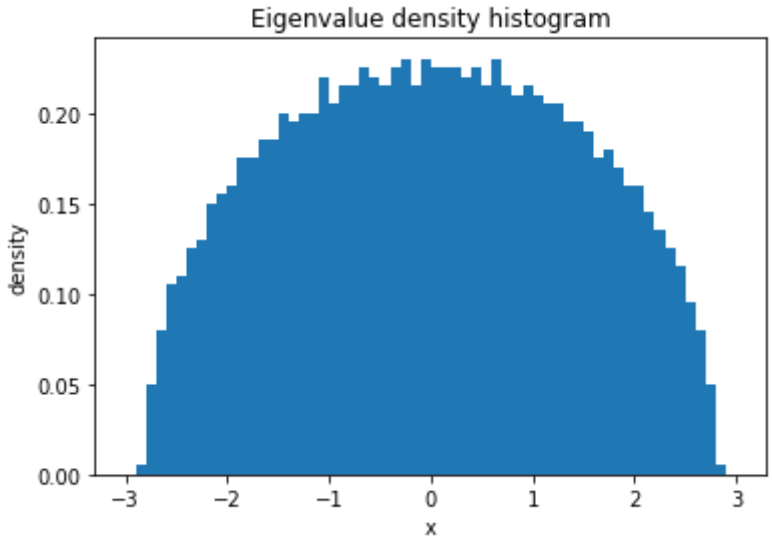
In [47]:

```
goe = GaussianEnsemble(beta=1, n=2000, use_tridiagonal=True)
goe.plot_eigval_hist(bins=60, interval=(-2,2), density=True)
```



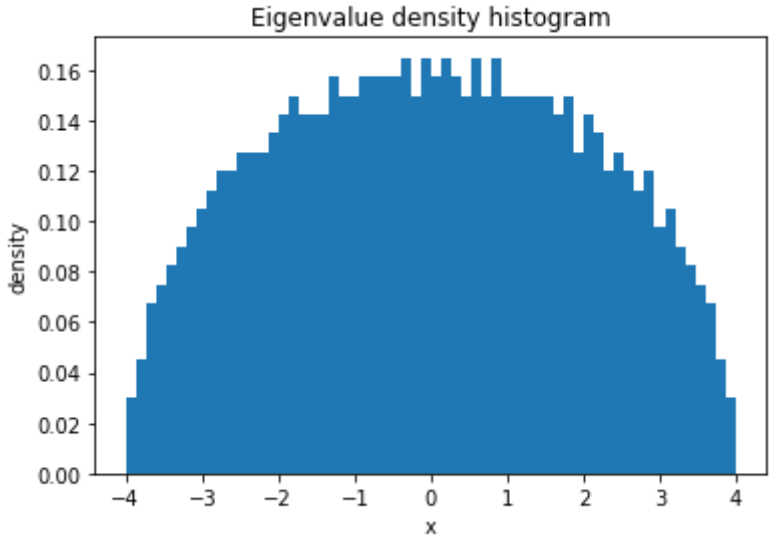
In [46]:

```
gue = GaussianEnsemble(beta=2, n=2000, use_tridiagonal=True)
gue.plot_eigval_hist(bins=60, interval=(-3,3), density=True)
```



In [43]:

```
gse = GaussianEnsemble(beta=4, n=1000, use_tridiagonal=True)
gse.plot_eigval_hist(bins=60, interval=(-4,4), density=True)
```

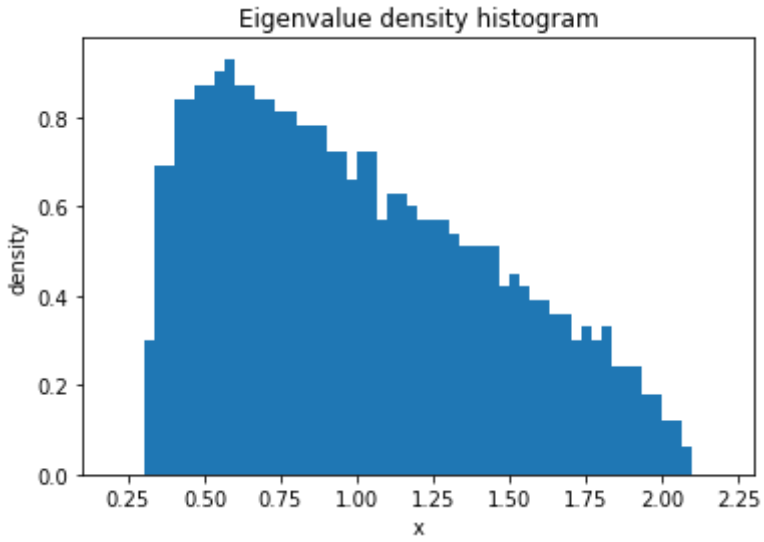


2.2 Wishart Ensemble

Wishart Ensemble eigenvalues follow Marchenko-Pastur Law.

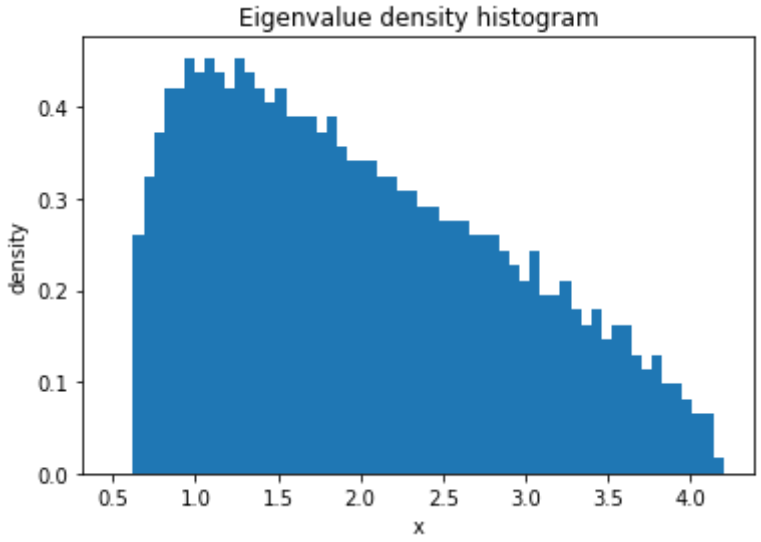
In [29]:

```
wre = WishartEnsemble(beta=1, p=1000, n=5000, use_tridiagonal=True)
wre.plot_eigval_hist(bins=60, interval=(0.2, 2.2), density=True)
```



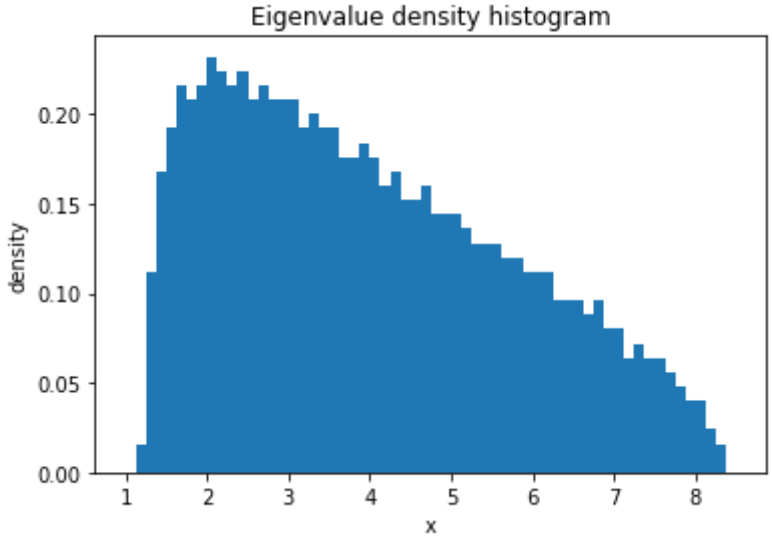
In [42]:

```
wce = WishartEnsemble(beta=2, p=1000, n=5000, use_tridiagonal=True)
wce.plot_eigval_hist(bins=60, interval=(0.5, 4.2), density=True)
```



In [59]:

```
wce = WishartEnsemble(beta=4, p=1000, n=5000, use_tridiagonal=True)
wce.plot_eigval_hist(bins=60, interval=(1, 8.5), density=True)
```



2.3 Manova Ensemble

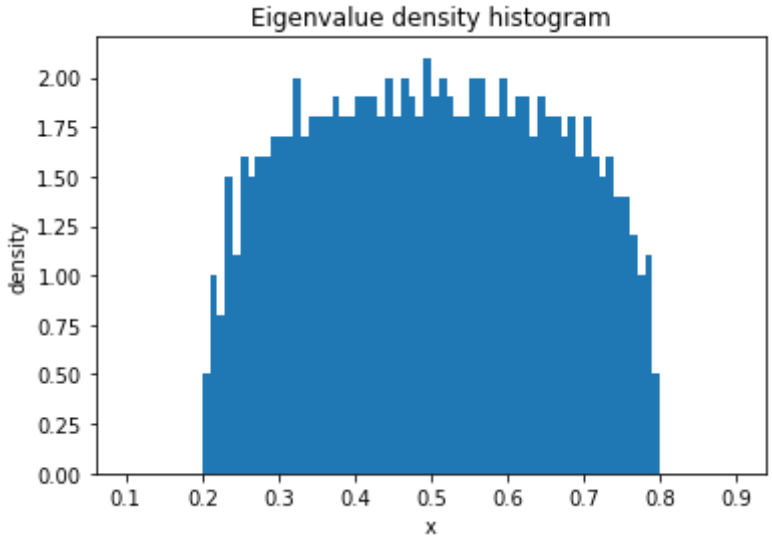
Depending on the sampled Manova Ensemble, its eigenvalues are going to be real or complex. The library implements both types of histograms (1D for real eigenvalues and 2D for complex eigenvalues).

In [60]:

```
m, n1, n2 = 1000, 5000, 5000
```

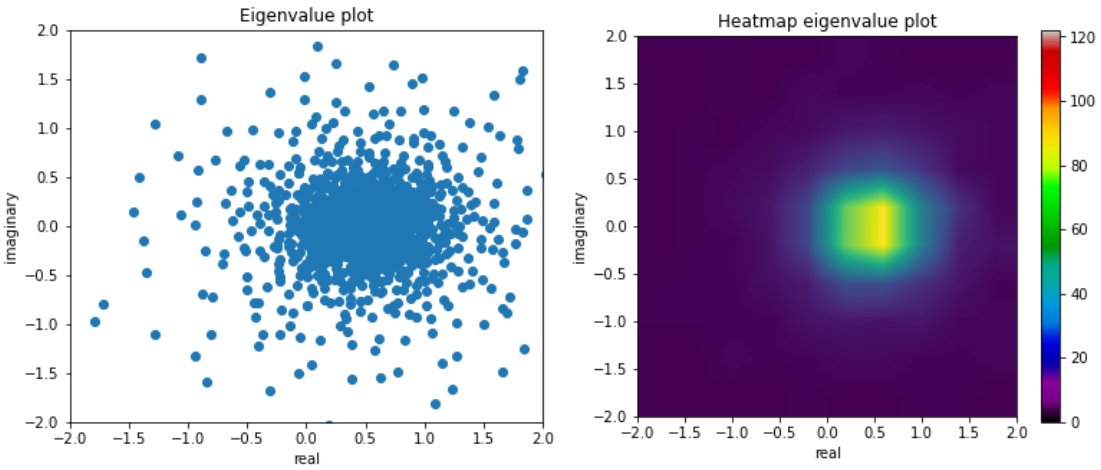
In [61]:

```
mre = ManovaEnsemble(beta=1, m=m, n1=n1, n2=n2)
mre.plot_eigval_hist(bins=80, interval=(0.1, 0.9), density=True)
```



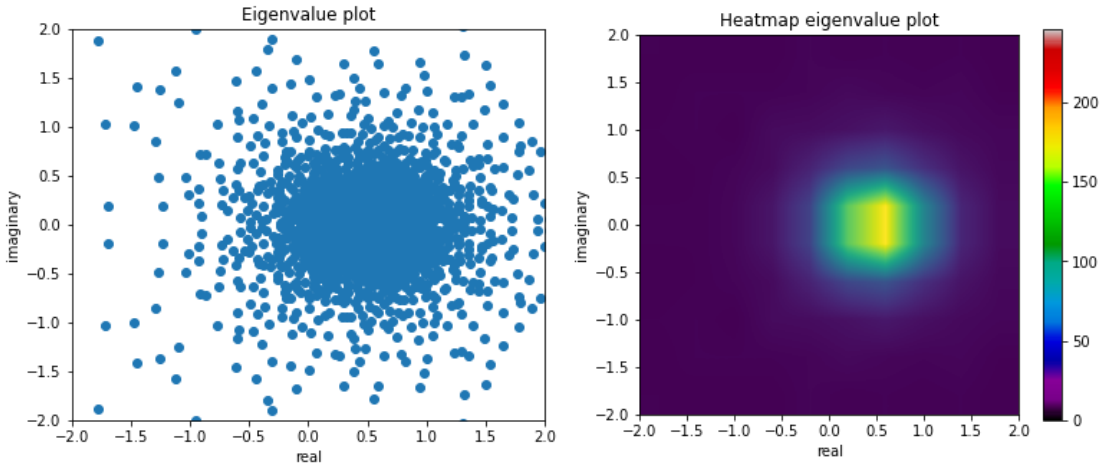
In [85]:

```
mce = ManovaEnsemble(beta=2, m=m, n1=n1, n2=n2)
mce.plot_eigval_hist(bins=80, interval=(-2,2))
```



In [84]:

```
mqe = ManovaEnsemble(beta=4, m=m, n1=n1, n2=n2)
mqe.plot_eigval_hist(bins=80, interval=(-2,2))
```

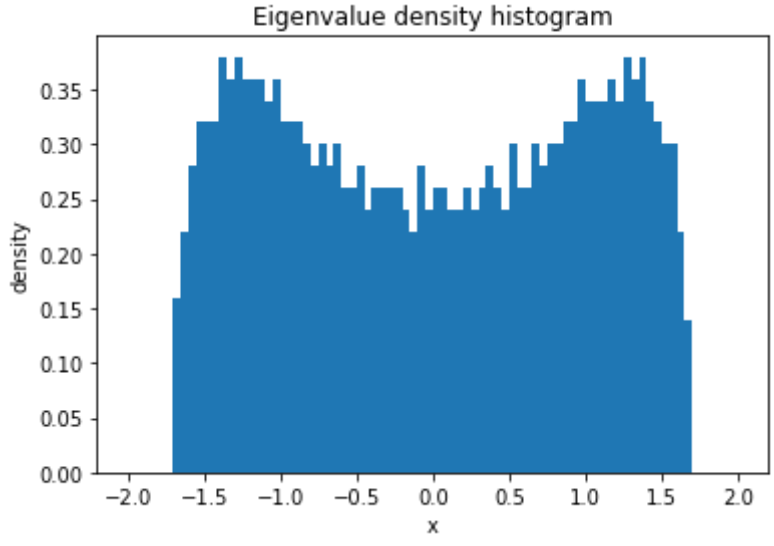


2.4 Circular Ensemble

Like Manova Ensemble, depending on the sampled Circular Ensemble, its eigenvalues are going to be real or complex. The library implements both types of histograms (1D for real eigenvalues and 2D for complex eigenvalues).

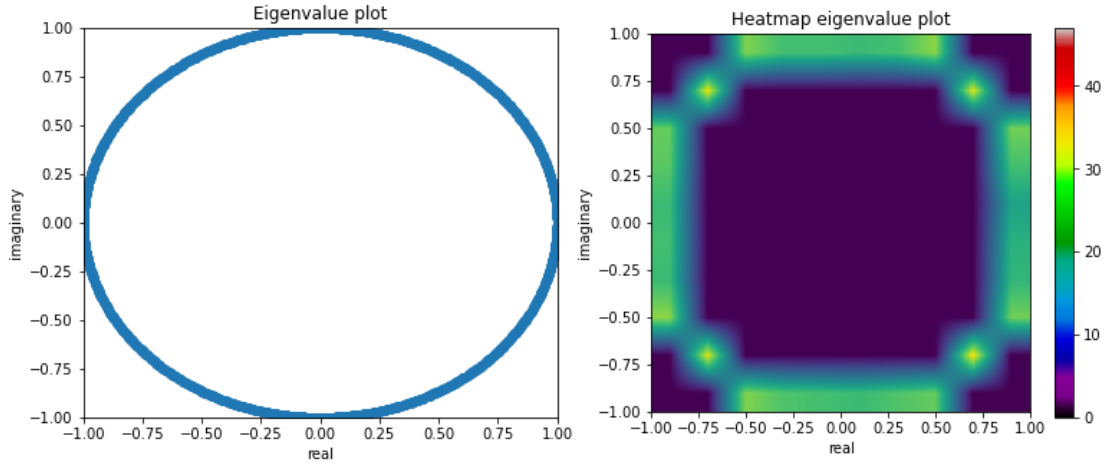
In [66]:

```
coe = CircularEnsemble(beta=1, n=1000)
coe.plot_eigval_hist(bins=80, interval=(-2,2), density=True)
```



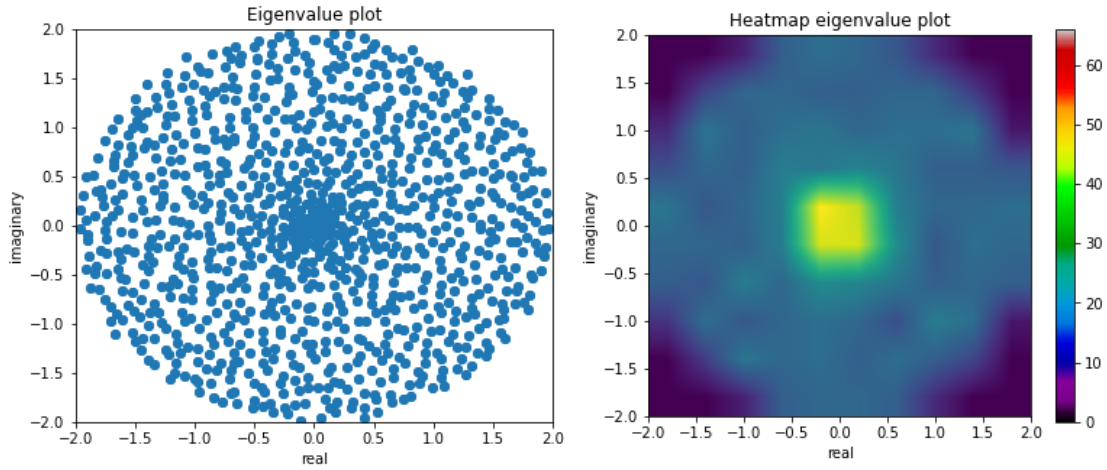
In [86]:

```
cue = CircularEnsemble(beta=2, n=1000)
cue.plot_eigval_hist(bins=80, interval=(-1,1))
```



In [87]:

```
cse = CircularEnsemble(beta=4, n=1000)
cse.plot_eigval_hist(bins=80, interval=(-2,2))
```



3. Comparing tridiagonal form vs standard form

In order to create the histograms of Gaussian Ensemble and Wishart Ensemble, their tridiagonal forms were used. Here we show the speed up of those routines.

In [72]:

```
t1 = time.time()
goe = GaussianEnsemble(beta=1, n=3000, use_tridiagonal=False)
hist = goe.eigval_hist(bins=60, interval=(-2,2))
t2 = time.time()

print('Required computation time (seconds):', t2-t1)
```

```
Required computation time (seconds): 4.547069787979126
```

In [73]:

```
t1 = time.time()
goe = GaussianEnsemble(beta=1, n=3000, use_tridiagonal=True)
hist = goe.eigval_hist(bins=60, interval=(-2,2))
t2 = time.time()

print('Required computation time (seconds):', t2-t1)
```

```
Required computation time (seconds): 0.3321700096130371
```

In [74]:

```
t1 = time.time()
wre = WishartEnsemble(beta=1, p=3000, n=5000, use_tridiagonal=False)
hist = wre.eigval_hist(bins=60, interval=(0,4))
t2 = time.time()

print('Required computation time (seconds):', t2-t1)
```

```
Required computation time (seconds): 5.298525094985962
```

In [75]:

```
t1 = time.time()
wre = WishartEnsemble(beta=1, p=3000, n=5000, use_tridiagonal=True)
hist = wre.eigval_hist(bins=60, interval=(0,4))
t2 = time.time()

print('Required computation time (seconds):', t2-t1)
```

```
Required computation time (seconds): 0.3530607223510742
```

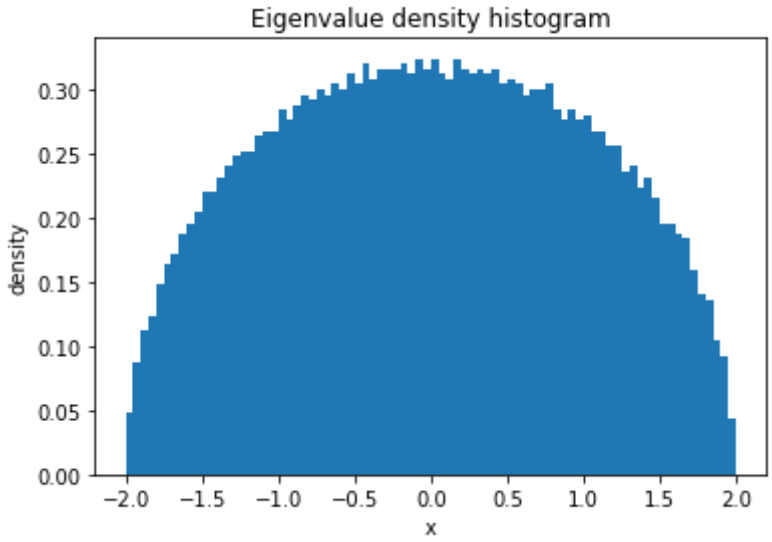
4. Spectral laws

We can directly study spectral Laws without needing to handle and sample random matrices directly.

4.1 Wigner's Semicircle Law

In [78]:

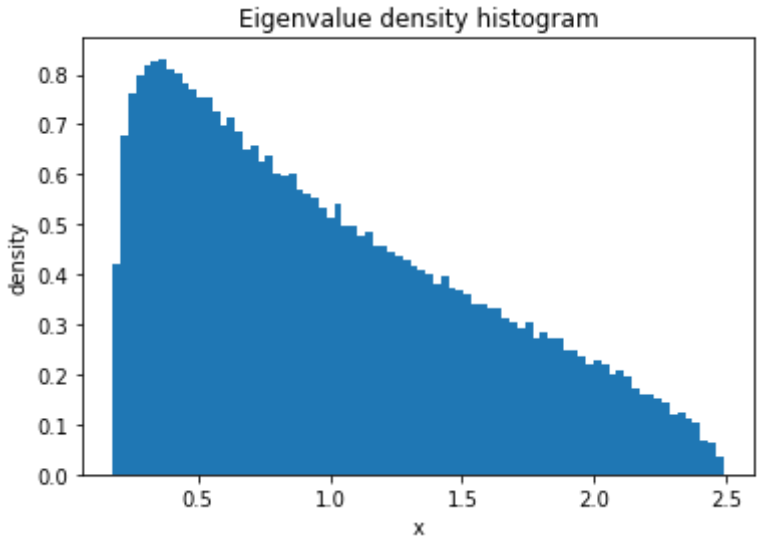
```
wigner_semicircular_law(ensemble='goe', n_size=5000, bins=80, density=True)
```



4.2 Marchenko-Pastur Law

In [79]:

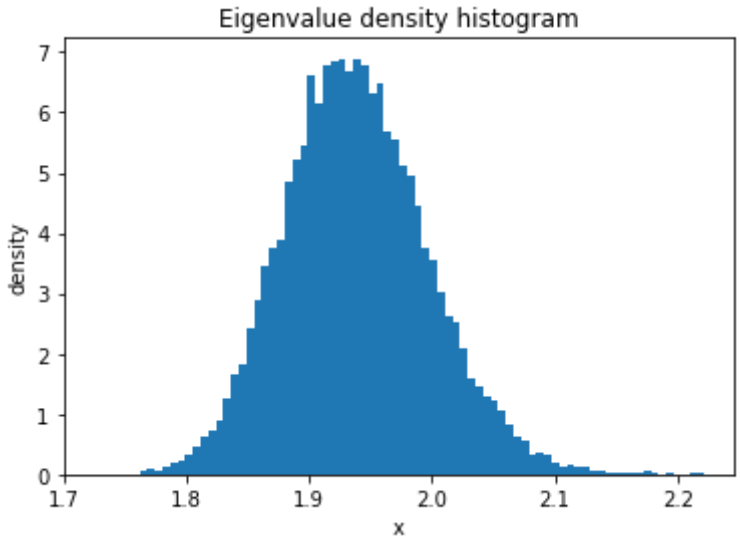
```
marchenko_pastur_law(ensemble='wre', p_size=5000, n_size=15000, bins=80, densit
```



4.3 Tracy-Widom Law

In [80]:

```
tracy_widom_law(ensemble='goe', n_size=100, times=20000, bins=80, density=True)
```



E

Comparación estimadores del módulo *covariance*

En este apéndice se muestra un *jupyter notebook* de ejemplo de cómo podríamos reproducir las simulaciones de la sección 5.4.

Algunas simulaciones son ejecutadas con funciones auxiliares, las cuales son expuestas en el código E.1 e incluidas en la librería en el directorio `./notebooks/tutorial_utils.py`.

Comparing *covariance* module estimators

1. Covariance estimators

Using certain metrics, we can simulate the behavior of implemented covariance matrix estimators.

In [103]:

```
# importing estimators
from covariance import sample_estimator, fsopt_estimator
from covariance import linear_shrinkage_estimator
from covariance import analytical_shrinkage_estimator
from covariance import empirical_bayesian_estimator
from covariance import minimax_estimator
# importing metrics
from covariance import loss_mv, prial_mv
# importing utils
from tutorial_utils import sample_pop_cov
from tutorial_utils import plot_cov_estimator_sim
```

In [89]:

```
p, n = 200, 600
values = [1, 3, 10]
prop = [0.2, 0.4, 0.4]
```

In [91]:

```
Sigma = sample_pop_cov(p, values, prop)
X = np.random.multivariate_normal(np.random.randn(p), Sigma, size=n)
print('Shape:', X.shape)
```

```
Shape: (600, 200)
```

In [96]:

```
Sigma_sample = sample_estimator(X)
# the printed shape should be equal to (p,p)
print('S sample estimator shape:', Sigma_sample.shape)
```

```
S sample estimator shape: (200, 200)
```

In [97]:

```
Sigma_fsopt = fsopt_estimator(X, Sigma)
# the printed shape should be equal to (p,p)
print('S sample estimator shape:', Sigma_fsopt.shape)
```

```
S sample estimator shape: (200, 200)
```

Checking that PRIAL(S) = 0%

In [98]:

```
exp_sample = loss_mv(sigma_tilde=Sigma_sample, sigma=Sigma)
exp_sigma_tilde = loss_mv(sigma_tilde=Sigma_sample, sigma=Sigma)
exp_fsopt = loss_mv(sigma_tilde=Sigma_fsopt, sigma=Sigma)

prial_mv(exp_sample=exp_sample, exp_sigma_tilde=exp_sigma_tilde, exp_fsopt=exp_
```

Out[98]:

```
0.0
```

Checking that PRIAL(S_fsopt) = 100%

In [99]:

```
exp_sample = loss_mv(sigma_tilde=Sigma_sample, sigma=Sigma)
exp_sigma_tilde = loss_mv(sigma_tilde=Sigma_fsopt, sigma=Sigma)
exp_fsopt = loss_mv(sigma_tilde=Sigma_fsopt, sigma=Sigma)

prial_mv(exp_sample=exp_sample, exp_sigma_tilde=exp_sigma_tilde, exp_fsopt=exp_
```

Out[99]:

```
1.0
```

1.1 Simulation increasing dimension *p*, but fixing ratio *p/n*

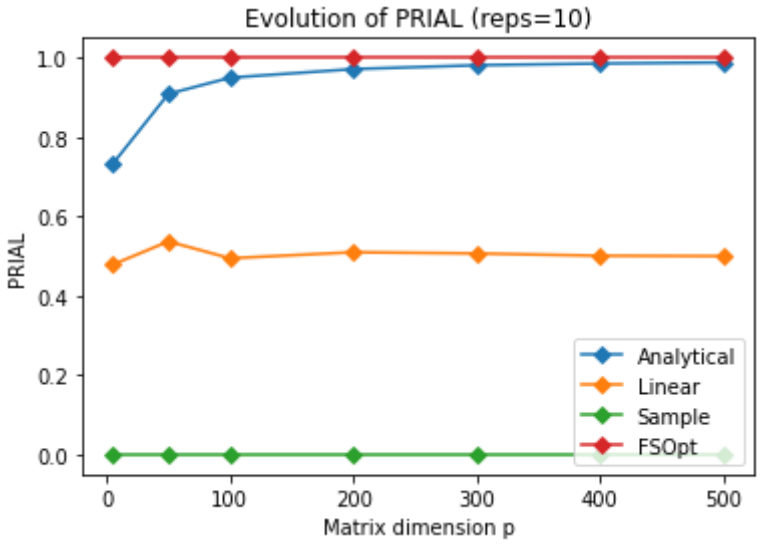
Using just non-linear shrinkage analytical estimator and linear shrinkage estimator (both asymptotically optimal)

In [108]:

```
estimators = [analytical_shrinkage_estimator, linear_shrinkage_estimator]

labels = ['Analytical', 'Linear']
P_list = [5, 50, 100, 200, 300, 400, 500]

plot_cov_estimator_sim(estimators, labels, values, prop, P_list=P_list, ratio=3
```



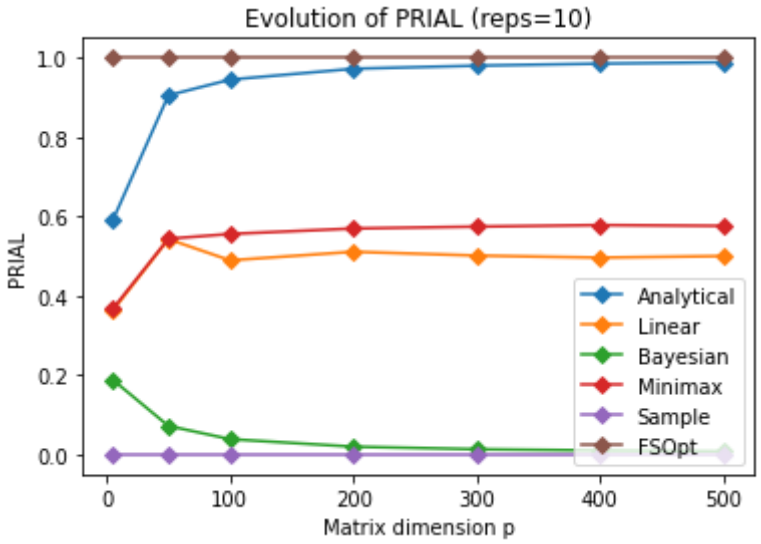
Using all estimators: adding empirical bayesian estimator and minimax estimator

In [111]:

```
estimators = [analytical_shrinkage_estimator, linear_shrinkage_estimator,
              empirical_bayesian_estimator, minimax_estimator]

labels = ['Analytical', 'Linear', 'Bayesian', 'Minimax']
P_list = [5, 50, 100, 200, 300, 400, 500]

plot_cov_estimator_sim(estimators, labels, values, prop, P_list=P_list, ratio=3
```



1.2 Simulation varying ratio *p/n*

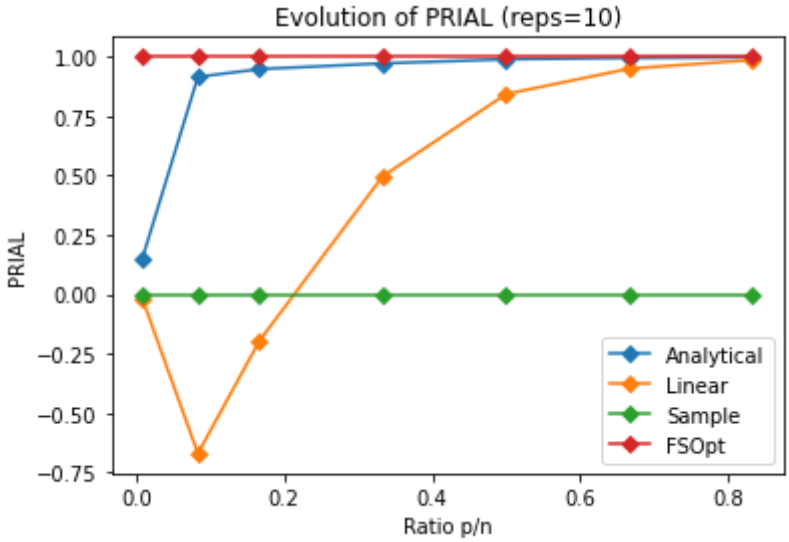
Using just non-linear shrinkage analytical estimator and linear shrinkage estimator (both asymptotically optimal)

In [112]:

```
estimators = [analytical_shrinkage_estimator, linear_shrinkage_estimator]

labels = ['Analytical', 'Linear']
P_list = [5, 50, 100, 200, 300, 400, 500]

plot_cov_estimator_sim(estimators, labels, values, prop, P_list=P_list, N=600,
```



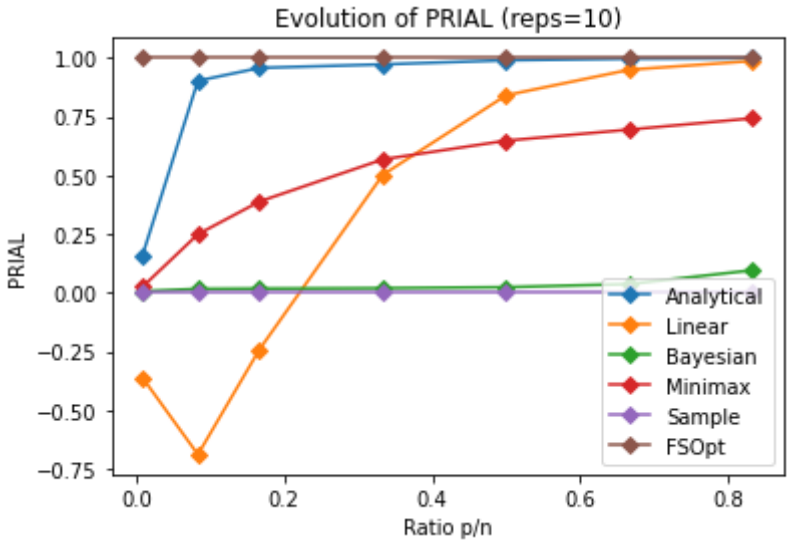
Using all estimators: adding empirical bayesian estimator and minimax estimator

In [113]:

```
estimators = [analytical_shrinkage_estimator, linear_shrinkage_estimator,
              empirical_bayesian_estimator, minimax_estimator]

labels = ['Analytical', 'Linear', 'Bayesian', 'Minimax']
P_list = [5, 50, 100, 200, 300, 400, 500]

plot_cov_estimator_sim(estimators, labels, values, prop, P_list=P_list, N=600,
```



1.3 Time simulation

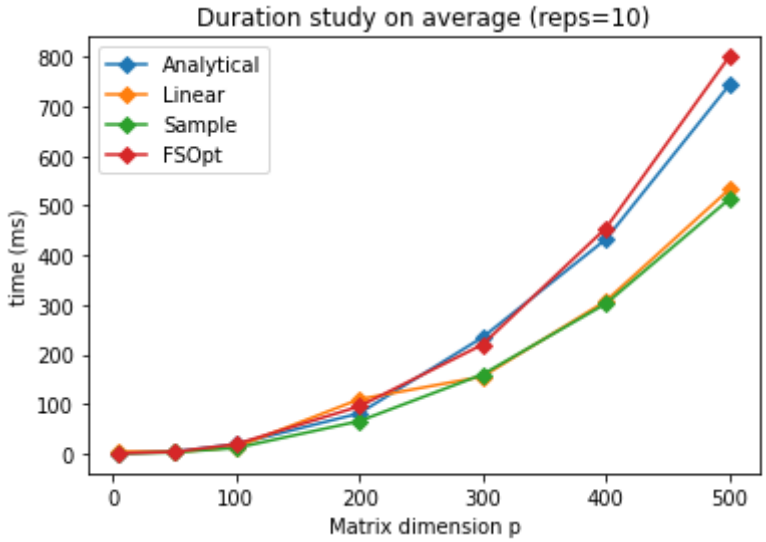
Using just non-linear shrinkage analytical estimator and linear shrinkage estimator (both asymptotically optimal)

In [119]:

```
estimators = [analytical_shrinkage_estimator, linear_shrinkage_estimator]

labels = ['Analytical', 'Linear']
P_list = [5, 50, 100, 200, 300, 400, 500]

plot_cov_estimator_sim(estimators, labels, values, prop, P_list=P_list, ratio=3
```



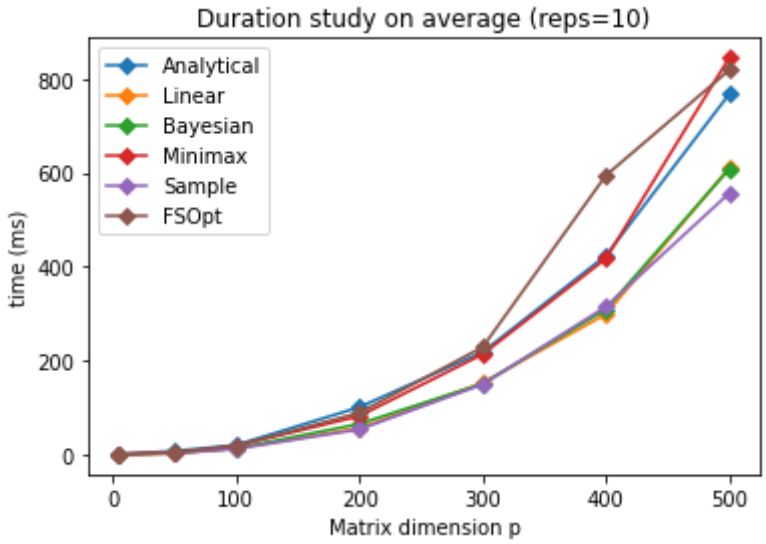
Using all estimators: adding empirical bayesian estimator and minimax estimator

In [120]:

```
estimators = [analytical_shrinkage_estimator, linear_shrinkage_estimator,
              empirical_bayesian_estimator, minimax_estimator]

labels = ['Analytical', 'Linear', 'Bayesian', 'Minimax']
P_list = [5, 50, 100, 200, 300, 400, 500]

plot_cov_estimator_sim(estimators, labels, values, prop, P_list=P_list, ratio=3
```



```
#Function to sample a random orthogonal matrix of size n times n
def sample_rand_orthogonal_mtx(n):
    # n by n random complex matrix
    X = np.random.randn(n,n)
    # orthonormalizing matrix using QR algorithm
    Q,_ = np.linalg.qr(X)
    return Q


#Function to sample a diagonal matrix with given values and proportions
def sample_diagEig_mtx(p, values, prop):
    n_per_val = []

    for i in range(len(values[:-1])):
        n_per_val.append(math.floor(p * prop[i]))
    n_per_val.append(p - np.sum(n_per_val))

    eigvals = []
    for (i,nval) in enumerate(n_per_val):
        eigvals += [values[i]]*nval

    # shuffling eigenvalues
    np.random.shuffle(eigvals)
    # building diagonal matrix
    M = np.diag(eigvals)
    return M


#Function to sample a certain population covariance matrix
def sample_pop_cov(p, values, prop, diag=False):
    if diag:
        return sample_diagEig_mtx(p, values, prop)
    else:
        O = sample_rand_orthogonal_mtx(p)
        M = sample_diagEig_mtx(p, values, prop)
        # O M O.T preserves original eigenvalues (O is an orthogonal rotation)
        return np.matmul(np.matmul(O, M), O.T) # sampling \Sigma


#Function to generate a dataset given a population covariance matrix
def sample_dataset(p, n, Sigma):
    X = np.random.multivariate_normal(np.random.randn(p), Sigma, size=n)
    return X


#Function that runs a covariance estimator simulation
def cov_estim_simulation(p, n, estimators, eigvals, props, nreps=100):
    # adviced to check prial_mv formula to understand the code below
    Sn_idx = 0
    Sstar_idx = 1
    Sigma_tilde_idx = 2
    # generating population covariance matrix
    Sigma = sample_pop_cov(p, eigvals, props)

    # matrices/arrays of results
```

```
    # +2 because sample and FSOptimal estimators are always considered
    LOSSES = np.zeros((len(estimators)+2, 3))
    PRIALS = np.zeros(len(estimators)+2)
    TIMES = np.zeros((len(estimators)+2))

    for (idx, estimator) in enumerate(estimators):
        t1 = time.time()
        for i in range(nreps):
            # sampling random dataset from fixed population covariance matrix
            X = sample_dataset(p=p, n=n, Sigma=Sigma)
            # estimating sample cov
            Sample = sample_estimator(X)
            # estimating S_star
            S_star = fsopt_estimator(X, Sigma)
            # estimating population covariance matrix using current estimator
            Sigma_tilde = estimator(X)
            # calculating losses
            loss_Sn = loss_mv(sigma_tilde=Sample, sigma=Sigma)
            loss_Sstar = loss_mv(sigma_tilde=S_star, sigma=Sigma)
            loss_Sigma_tilde = loss_mv(sigma_tilde=Sigma_tilde, sigma=Sigma)
            LOSSES[idx][Sn_idx] += loss_Sn
            LOSSES[idx][Sstar_idx] += loss_Sstar
            LOSSES[idx][Sigma_tilde_idx] += loss_Sigma_tilde
        t2 = time.time()
        TIMES[idx] = (t2-t1)*1000/nreps # time needed in ms (meaned by number of repetitions)
        LOSSES[idx] /= p
        PRIALS[idx] = prial_mv(exp_sample=LOSSES[idx][Sn_idx],
                               exp_sigma_tilde=LOSSES[idx][Sigma_tilde_idx],
                               exp_fsopt=LOSSES[idx][Sstar_idx])

    # Sample estimator
    t1 = time.time()
    for i in range(nreps):
        # sampling random dataset from fixed population covariance matrix
        X = sample_dataset(p=p, n=n, Sigma=Sigma)
        # estimating sample cov
        Sample = sample_estimator(X)
        # estimating S_star
        S_star = fsopt_estimator(X, Sigma)
        # estimating population covariance matrix using sample estimator
        Sigma_tilde = sample_estimator(X)
        # calculating losses
        loss_Sn = loss_mv(sigma_tilde=Sample, sigma=Sigma)
        loss_Sstar = loss_mv(sigma_tilde=S_star, sigma=Sigma)
        loss_Sigma_tilde = loss_mv(sigma_tilde=Sigma_tilde, sigma=Sigma)
        LOSSES[-2][Sn_idx] += loss_Sn
        LOSSES[-2][Sstar_idx] += loss_Sstar
        LOSSES[-2][Sigma_tilde_idx] += loss_Sigma_tilde
    t2 = time.time()
    TIMES[-2] = (t2-t1)*1000/nreps # time needed in ms (meaned by number of repetitions)
    LOSSES[-2] /= p
    PRIALS[-2] = prial_mv(exp_sample=LOSSES[-2][Sn_idx],
                          exp_sigma_tilde=LOSSES[-2][Sigma_tilde_idx],
                          exp_fsopt=LOSSES[-2][Sstar_idx])

    # FSOpt estimator
```

```
    t1 = time.time()
    for i in range(nreps):
        # sampling random dataset from fixed population covariance matrix
        X = sample_dataset(p=p, n=n, Sigma=Sigma)
        # estimating sample cov
        Sample = sample_estimator(X)
        # estimating S_star
        S_star = fsopt_estimator(X, Sigma)
        # estimating population covariance matrix using current estimator
        Sigma_tilde = fsopt_estimator(X, Sigma)
        # calculating losses
        loss_Sn = loss_mv(sigma_tilde=Sample, sigma=Sigma)
        loss_Sstar = loss_mv(sigma_tilde=S_star, sigma=Sigma)
        loss_Sigma_tilde = loss_mv(sigma_tilde=Sigma_tilde, sigma=Sigma)
        LOSSES[-1][Sn_idx] += loss_Sn
        LOSSES[-1][Sstar_idx] += loss_Sstar
        LOSSES[-1][Sigma_tilde_idx] += loss_Sigma_tilde
    t2 = time.time()
    TIMES[-1] = (t2-t1)*1000/nreps # time needed in ms (meaned by number of repetitions)
    LOSSES[-1] /= p
    PRIALS[-1] = prial_mv(exp_sample=LOSSES[-1][Sn_idx],
                          exp_sigma_tilde=LOSSES[-1][Sigma_tilde_idx],
                          exp_fsopt=LOSSES[-1][Sstar_idx])

    return LOSSES, PRIALS, TIMES


#Function that runs a covariance estimator simulation and plots the result
def plot_cov_estimator_sim(estimators, labels, eigvals, props, P_list,
                           N=None, ratio=3, nreps=None, metric='prial'):

    # +2 because Sample and FSOptimal estimators are always considered
    MEASURES = np.zeros((len(P_list), len(estimators)+2))
    labels += ['Sample', 'FSOpt']

    ratios = []

    for (idx, p) in enumerate(P_list):
        if N is None:
            n = ratio*p
        else:
            n = N
            ratios.append(p/n)
        if nreps is None:
            nreps = int(max(100, min(1000, 10000/p)))

        losses,prials,times = cov_estim_simulation(p, n, estimators, eigvals, props, nreps=nreps)
        if metric == 'prial':
            MEASURES[idx] = prials
        elif metric == 'loss':
            MEASURES[idx] = losses
        elif metric == 'time':
            MEASURES[idx] = times

    if N is None:
        lines = plt.plot(P_list, MEASURES, '-D')
```

```
169         plt.xlabel('Matrix dimension p')
170     else:
171         lines = plt.plot(ratios, MEASURES, '-D')
172         plt.xlabel('Ratio p/n')
173     plt.legend(lines, labels)
174 
175     if metric == 'prial':
176         plt.title('Evolution of PRIAL (reps='+str(nreps)+')')
177         plt.ylabel('PRIAL')
178     elif metric == 'loss':
179         plt.title('Evolution of Loss (reps='+str(nreps)+')')
180         plt.ylabel('Loss')
181     elif metric == 'time':
182         plt.title('Duration study on average (reps='+str(nreps)+')')
183         plt.ylabel('time (ms)')
```

Código E.1: Funciones útiles para las simulaciones de evaluación de estimadores de matrices de covarianza